USENIX Association

Proceedings of the

USENIX Conference

on

Object-Oriented Technologies

(COOTS)

June 26-29, 1995

Monterey, California, USA

Table of Contents

USENIX Conference on Object-Oriented Technologies (COOTS)

June 26-29, 1995
Monterey, California

Wednesday, June 28, 1995

Thursday, June 29, 1995

Distributed Object Systems II
Session Chair: Jim Waldo

Object Potpourri
Session Chair: Christopher Pettus

CONFERENCE ORGANIZERS

Technical Sessions Program Chair: *Vincent F. Russo, Purdue University*
Tutorial Program Chair: *Doug Lea, SUNY Oswego*

Program Committee:
Luca Cardelli, Digital Systems Research Center
Murthy Devarokonda, IBM T. J. Watson Research Center
Ted Goldstein, Sun Microsystems Laboratories
Paul Leach, Microsoft
Mark Linton, Silicon Graphics, Inc.
Christopher Pettus, Taligent
Jim Waldo, Sun Microsystems Laboratories

Tutorial Program Coordinator: *Daniel V. Klein, USENIX*
Conference Planner: *Judith F. DesHarnais, USENIX*

Simple Activation for Distributed Objects

Ann Wollrath, Geoff Wyant, and Jim Waldo
Sun Microsystems Laboratories
{ann.wollrath, geoff.wyant, jim.waldo}@east.sun.com

Abstract

In order to support long-lived distributed objects, object activation is required. Activation allows an object to alternate between periods of activity, where the object implementation executes in a process; and periods of dormancy, where the object is on disk and utilizes no system resources.

We describe an activation protocol for distributed object systems. The protocol features overall simplicity as well as applicability to several different activation models. We use the Modula-3 network object system as a base for our implementation; while we make no changes to the underlying network object subsystem, we suggest a minor modification that could be made to the marshalling of network objects to assist in lazy activation, our preferred activation model.

1 Introduction

Distributed object systems are designed to support long-lived persistent objects. Given that these systems will be made up of many thousands (perhaps millions) of such objects, it would be unreasonable for object implementations to become active and remain active, taking up valuable system resources, for indefinite periods of time. In addition, clients need the ability to store persistent references to objects so that communication among objects can be re-established after a system crash, since typically a reference to a distributed object is valid only while the object is active.

Such systems need to provide *activation*, a mechanism for providing persistent references to objects and managing the execution of object implementations. When warranted, object servers can be started up or shut down.

As our platform for distributed objects, we use the network object system [1] provided with the Digital Equipment Corporation System Research Center distribution of the Modula-3 (M3) language [2]. Given that this implementation platform does not support object activation, we have designed an activation protocol to address that need. We identify several important goals for our activation protocol:

- flexible mechanism enabling implementations of a variety of activation models;
- simple activation scheme focused on the core protocol leaving special cases of activation to be specified at a higher layer of abstraction;
- minimal implementation requirements on servers supporting the protocol.

Our implementation of the protocol is constrained by following components of the underlying system:

- the need to be layered on top of the network object system without requiring changes to the network object runtime;
- the protocol should require no modification to existing network object stub generators.

Other factors have influenced the design of the interfaces (more cosmetically):

- interfaces are restricted to single inheritance (although this restriction did not interfere with the design since multiple inheritance was ultimately not required);
- interfaces reflect Modula-3 naming conventions (we have adopted the convention of naming the primary interface in any module `T`, following Modula-3 style definitions).

The activation protocol is specified using the Object Management Group's Interface Definition Language (OMG IDL) [3]. In our protocol definition we use an additional keyword, *serverless*, which denotes a pass-by-value object (i.e., library objects). The serverless feature is not part of standard IDL, but could be thought of (loosely) as an IDL `struct`.

The notion of pass-by-value objects is an attractive one, since exposing a data definition explicitly via `struct` violates data encapsulation. A client possessing a structure may manipulate that structure without restriction, potentially altering the data in a manner not intended or expected by a subsequent

recipient. By using a pass-by-value (or serverless) object instead, data can be hidden and a more appropriate interface can control access to and modification of that data. We include serverless objects for just this reason.

The obvious drawback to pass-by-value objects in distributed systems is that the code for the library object implementation must be linked into all clients of such an object. This requirement is much different than having to link in stubs (surrogate code) for a distributed object; all clients must link to the *same* library object implementation. Clients that transmit serverless objects that have a different implementation than the receiver expects will likely cause unanticipated failure: a failure usually encountered during the unmarshalling process performed at the receiver.

2 Activation Models

In a distributed system in which the total number of objects that could be used exceeds the total system resources available, some way of conserving system resource needs to be found. One way of conserving resources is to distinguish between *active* objects, which take up system resources, and *passive* objects, which do not.

More precisely, an *active* object is one that is associated with a process on some system. A *passive* object is one that is not associated with such a process, but which can be brought into an active state. Transforming a passive object into an active object is a process we refer to as *activation*. Activation requires that an object be associated with a process, which may entail loading the code for that object into a process and restoring any persistent state for the object.

In this section, we describe the client's model of activation, followed by a discussion of various activation models and the trade-offs associated with each.

We will refer to the activation models that we will discuss as *eager* (or deep) activation, *lazy* activation, and *split* activation. Eager activation can be characterized as a strategy that maintains as an invariant that each reference within an active object also refers to an active object. Lazy activation, in contrast, is a model that defers activation of an object to the time at which an operation is invoked on that object. Split activation combines aspects of both schemes.

Client Model

Our basic model, from the client side, involves two distinct forms of reference for objects that are (potentially) in a separate address space. The first of these, called the *internal* reference, typically points to a local surrogate or proxy for an object. From the programmatic point of view, such internal references look like references to other, local objects. These internal references are the entities that are manipulated, used for method invocations, and passed around to other objects. In our system, internal references are simple object references that derive from the base class of NetObj.T, the Modula-3 Network Object class.

Internal references in this framework, however, are not guaranteed to refer to a valid object over various runs of the code that implements that object. In particular, if an object is made passive (or crashes and is restarted) the internal reference for it may change. If a client wishes to store a reference to a (potentially) remote object as part of the persistent state of that client, some form of reference is needed that will survive those changes. We refer to this form of reference as an *external* reference.

When a client has an external reference to an object, that reference needs to be converted to an internal reference if the client wishes to make use of the object. Such a conversion can require the activation of the referred to object.

Eager Activation

Eager (or deep) activation denotes an activation model whereby an object, and the objects reachable from that object, are activated at once. In this model when a client restores its state, all external object references are presented as internal object references. As part of restoring an internal reference from its external form, the referred-to object is made active. Additionally, the object's implementation restores its state, causing activation of those objects denoted by external references. Thus, when activating a single object, the transitive closure of objects referenced by that specific object is activated.

This model has the advantage that it requires minimal intrusion on the client-side. A small amount of generic code needs to be written that converts object references from their external form to their internal form and to call the appropriate object activation service. This model also has the advantage that it can be determined at the time the object reference is converted to its internal form whether or not that object can be made available.

One primary disadvantage of this model is that it doesn't handle circular chains of reference. Circular chains of reference occur when an object refers back to itself through any number of intermediate references (e.g., A contains a reference to B which contains a reference back to A). An object can also contain a self-reference. In the eager model described above, activating an object containing a reference that eventually refers back to itself (a circular reference) requires activating the object itself and can lead to deadlock unless complicated avoidance mechanisms are employed.

Another problem with this model concerns scalability. In the eager model, an object is activated when its external reference is read from disk rather than when it is first dereferenced. This strategy seriously affects the scalability of activation. Upon reading an object's persistent state, a potentially large number of objects may be activated at once if the object contains many external object references. Activating an entire tree of objects can cause an "activation storm" where cascading activation requests flood the system.

Another disadvantage to this eager approach is that it will activate an object, even if the object is never used by the client. Ideally, unused object should not consume any system resources. Consider a object which contains references to 100 other objects on 100 other machines. When that object restores these references from disk, 100 processes will be created in an eager fashion regardless of whether the references are actually used by the object. Such "storms" seriously effect the overall performance and predictability of the system.

Lazy Activation

Lazy activation defers activating an object until a client's first use (i.e. the first method invocation). This model of activation is typically implemented by generating stubs that check to see if the target object has been made active for this process. We refer to these stubs as *fault blocks*. Each fault block maintains a reference to the target object. If this reference is nil, the target has not been activated for this process. Upon method invocation, the fault block (for that object) then engages in the activation protocol and retains the reference to the newly activated object. From that point on, all fault block stubs forward method invocations to the surrogate object that stands in for the remote object. This scheme is analogous to "object faulting" in persistent object systems [4] or page faulting upon referencing non-resident memory locations.

The lazy activation approach avoids the deadlock problem of eager activation. Except for certain pathological cases, activation of an object never requires the activation of itself even in the case of circular references.

This scheme is also more scalable. Since activation is deferred until the time of first reference, an object is never activated unless explicitly used. Thus, needless process creation (for unused objects) is eliminated as are activation storms.

A major drawback of this approach is that notification that an object is not available will not occur until the first invocation on the object. At this point, the client may have no sensible recovery strategy and its only option is to exit.

Another drawback is that two stubs exist on the client side for each non-local object. The first is the surrogate that performs remote method invocation. The second is the "fault block" which determines the activation status of the object and activates it if needed and forwards invocations to the surrogate.

One must be careful not to expose fault blocks to the client. This exposure can lead to subtle failures since the client is operating under the assumption that it holds a true reference to the object—one that can be passed as an argument or upon which operations can be invoked—not a reference to a fault block.

For the system to function properly, the functionality of fault blocks must be part of the surrogate (or handled somewhere in the runtime). Due to our desire not to modify the network object runtime or stub (surrogate) generators, we chose not to implement lazy activation at this time. Clean integration of lazy activation with the existing system requires modification to marshalling surrogate objects and to the runtime itself.

Split Activation

We chose to use the hybrid model of split activation. The split activation model is one in which the responsibility for activation is divided into two parts. The first part activates a process for the object. The second part activates the object state on the first invocation of that object. Process activation occurs when the object reference is converted from its external form to its internal form. State activation occurs upon first use of the object.

The split model can be implemented by server-side fault blocks that perform an analogous function to the client-side fault blocks used in lazy activation. In this scheme, a server-side fault block checks to see if the

target object is active. If not, it activates the object's state (via a server-provided callback). From that point on, the fault block forwards method invocations to the active object. Split activation sits between eager and lazy activation. While this approach does require special handling on the server side, it does not require any changes to the runtime system.

This model avoids the deadlock problem of eager activation. State activation does not occur until first use of the object, thus breaking potential activation cycles. It also avoids the multiple-object problem that the lazy activation approach suffers from. Clients never deal with two forms of the object: the fault block vs. the surrogate. Clients only deal with surrogates.

This scheme also has the advantage that the server can determine at process activation time whether or not the target object can be reached. Thus, clients can learn earlier about the availability of an object than could be done in the lazy activation approach. A client may be in a better state to recover from this situation. Note that, at any point in the future, communication with an object may fail. A client must be able to attempt recovery from this potential occurrence as well. In split activation, while a client of an object has the potential for early knowledge of failure, it does not necessarily mean that the client will be in the *best* state to handle such failure.

There are some disadvantages of this approach. It can be potentially less scalable then lazy activation if servers are not implemented with activation in mind. If the server restores a large set of object references, then a process will be created for each of those objects causing an "activation storm". However, this process will not propagate beyond the first level of a tree. A child won't propagate the activation until its state is needed. Again, this is upon first invocation. Upon first invocation, the child will restore its state. Any non-local object references will cause the next level of the tree/graph to have processes created for objects (though no state will be activated). Thus process activation occurs as a wavefront, but state activation occurs on demand.

The split activation model transfers control of when activation occurs from the runtime system to the server. A server can choose which objects are activated by restoring its persistent state selectively. We have designed a generic container class to deal with such selective activation of objects. Operations that access elements of the container handle the machinery of activation, thus emulating lazy activation for those objects in the container. Using the container abstraction allows the server to delegate the manual control of activation.

3 The Basic Activation Protocol

This section describes the basic activation protocol in detail. The protocol involves three entities: a client, an activator, and a server process/object. The activation protocol proceeds as follows (starting with the client):

1. *Obtain an external object reference.* Let's say a client wishes to retain a reference to an object for some period of time and to be able to store that reference on disk. From a name service or the object itself (using an internal reference to the object), the client obtains an external reference for that object. The abstraction for external object references will be discussed in the next subsection.

2. *Request activation.* At a later time, this external object reference (obtained previously) can be converted into an internal reference using an *activator*, on the same host as the referenced object, as facilitator. An activator process (daemon) runs on each host in the system. It is the responsibility of the activator to spawn servers (on the local host) corresponding to certain external references. The client hands the activator an external reference together with a request to "activate" that object.

3. *Spawn server process.* Upon receiving the client's request for activation, the activator spawns a server process for the object, passing it relevant data (from the external reference) for bootstrapping purposes.

4. *Server activates.* The server process starts up and sends the internal reference for the object back to the activator, thus informing the activator that the object's activation is complete.

5. *Reply with internal reference to client.* The activator replies to the client with the internal reference that it received from the server process. The client is free to invoke operations on the activated object, using its internal (programmatic) reference.

The client mentioned in the above protocol does not necessarily denote the client application program. An ideal implementation would completely hide the mechanisms of activation from the client.

Activation Interfaces

The interfaces that embody the activation protocol described above are contained in the module Activation defined in IDL (see Figure 1).

```
#include "VantageID.idl"
#include "VantageObj.idl"

module Activation {

   exception UnableToActivate {};
   exception AlreadyActive {};
   exception UnableToDeactivate {};
   exception AlreadyManaged {};
   exception WrongActivator {};

   typedef VantageID::T ID_T;

   interface Address_T : serverless {
     string toText();
     boolean equal(in Address_T addr);
   };

   interface Token_T : serverless {
     VantageID::T vid ();
     string executableFileName ();
     string activationData ();
     Address_T activatorAddress ();
     boolean isManaged ();
     string toText ();
   };

   interface Activatable_T :
              VantageObj::T {
     Token_T activationToken ();
   };

   interface Manager_T {
     Activatable_T activate (
          in Token_T token,
          in ID_T activationID);
   };

   interface Activator_T {
     Address_T address ();
     Activatable_T activate (
          in Token_T token)
       raises (UnableToActivate,
                WrongActivator);
     void managed (
          in Manager_T manager,
          in string executableFileName)
       raises (AlreadyManaged);
     ID_T activated (
          in Activatable_T activatedObj,
          in Token_T token)
       raises (AlreadyActive,
                WrongActivator);
     boolean isActive(
          in Token_T token)
       raises (WrongActivator);
     void deactivated (
          in ID_T id,
          in Token_T token)
       raises (UnableToDeactivate,
                WrongActivator);
   };
};
```

Figure 1. Module Activation

The Activation module refers to two other interface definition files. The interface described in the file "VantageObj.idl" is one supported by all objects in the overall system we are constructing. This interface, VantageObj::T, contains a single method, which returns an identifier that, with a high degree of probability, uniquely identifies the object. The interface VantageID::T, contained in the file "VantageID.idl", defines these unique identifiers for objects.

The major interfaces in the Activation module are:

- Address_T - abstraction for object location (pass-by-value),
- Token_T - abstraction for external object references (also pass-by-value),
- Activatable_T - interface supported by those objects capable of being activated,
- Activator_T - interface to the activator,
- Manager_T - interface for object servers that wish to support multiple objects per server process.

The following subsections will describe each of these interfaces in turn.

External Object References

An external reference to an object must contain enough information so that some mechanism can initiate the object's execution. That is, given some

form of external object reference, a client can obtain an internal reference to an active object.

The minimum information needed to initiate a server process is the location of the server program (i.e., executable) and the physical host location for the spawned server process. Additionally, the server may need some bootstrap information in order to read its persistent state, or export itself to a name service, for example.

In our system, external references are known as *activation tokens* (represented by the Token_T interface). An activation token (or simply a token) can be thought of as the meta-data for an object. These tokens must supply the following information:

- a unique identifier for an object,
- the *address* of the object's activator,
- the executable file name of the server program,
- server-specified activation data, and
- a flag indicating whether this object is *managed*.

Each object in our system exports a unique identifier (also referred to as a *vantage ID* or *VID*) which is used for object identification purposes by both the activator and the spawned server itself. We use these probabilistically unique identifiers in order to determine object equality, since reference equality for distributed objects is not supported in most systems. The unique identifier in the activation token, obtained via the vid operation, corresponds to the object's unique identifier.

The address of the activator, obtained as a result of the activatorAddress operation, is the abstract "location" of the activator and is implementation specific. The abstraction for an address is the Address_T interface which contains two operations toText (for converting the abstract representation to string form) and equal (for establishing equivalence of addresses). In our system, an activator address essentially refers to the activator's host.

Note that both Address_T and Token_T types are denoted by the serverless keyword. Both types are represented as pass-by-value objects as opposed to full-fledged distributed objects which would carry the overhead of remote method invocation for invoking each of their operations. Addresses and activation tokens must be light weight entities. Many implementation difficulties would arise if these objects were to be distributed in nature (that is, pass-by-reference).

The executable file name of the server program is the pathname to the server executable (returned via the executableFileName operation). The activator uses this information in the activation token to spawn a server for the object.

The activationData contained in the activation token is entirely under the control of the object (server) for which the token is created. It is up to the object implementation to decide what information is relevant to start-up, and place that information in the token. For example, the activation data for a server might be a pathname to a log file for recovery.

The isManaged flag simply indicates whether an object should share the same server process as other objects on the same node which share the same executable file name. In a later section, we explain our scheme for managing multiple objects within a single server.

In our system, activation tokens are self-contained entities. That is, the activator needs only the information present in a token to spawn a server process. An alternative design (and potentially more flexible) would be for an activation token to refer implicitly to a database containing activation information for objects. For example, a unique identifier would be the only required component of an activation token if the client consulted a database to determine the activator for an object, and each activator consulted a database for further information on the activation attributes of an object (such as its executable file name).

While this design may be more flexible—since it does not fix the contents of activation tokens—it does require that each activator have access to state information, i.e., the database of activation information. This introduces more complexity into the activator and additional failure modes during activation if databases become temporarily inaccessible. Management of such databases (registration of information, update, and querying) also increases the overhead of the activation mechanism.

Note that the address of the activator is embedded in the activation token. Since activation tokens can be stored persistently, an activation token for an object must not change over time, unless all objects to which the activation token was given can be contacted with the value of the new token for the object. Thus, object mobility cannot be easily employed since the contents of activation tokens are fixed at creation time. Other mechanisms must be built to handle moving objects in the system. Important services that are likely to relocate in the future may be stored in a name server and referenced by name rather than strictly by token.

Server Requirements

For an object to be capable of being activated, it must support the Activatable_T interface. Such an "activatable" object need only implement one operation, activationToken, that constructs an activation token containing the appropriate meta-data for the object. As described above, this token contains the object's identifier, executable file name, bootstrap data (activation data), and the address of the object's local activator; an activatable object can query the local activator (which should be a well-known entity) for its address for inclusion in the activation token.

It is the responsibility of each activatable object to hand out activation tokens to those objects requesting the information. Also, an activatable object is responsible for informing the activator when it has completed activation.

The simplest implementation of a server is a single object per server process. In our system, we have the additional notion of *aggregate* objects. An *aggregate* object has an object identity spanning multiple objects (in the same address space), but which functions as a single cohesive unit. Objects in the aggregate share the same identity and cooperate to implement the object's interface. All objects which share the same unique identifier (VID) are considered part of the same aggregate object. If the aggregate for a particular object identifier is "activatable", then all objects which make up the aggregate must implement the Activation::Activatable_T interface.

A server started up by the activator is passed several arguments:

```
-vid <ID> -activationData <data>
```

When the server starts up, it parses the arguments, activates itself, and informs the activator that it is activated. In the process of "activating itself" the server must be careful not to block. In an implementation of a lazy model of activation, blocking might be less of a problem. However, blocking still is a potential hazard if the object attempts to contact any other objects (for example during the server recovery process) before replying "activated" to the activator. Therefore, the server must invoke the activated operation on the activator as soon as possible, and before blocking, so that it will not hold up the client requesting activation or cause potential deadlock.

For aggregate objects, all objects within the aggregate can be activated at once by invoking the activated method on the activator for each activatable object in the aggregate (each having different activation data).

To avoid potential race conditions between servers starting up by hand, or being started via the activator, a server that receives the exceptional return AlreadyActive from the activated operation should exit immediately. This exception means that two servers with the same identity were either started in parallel or a server was started mistakenly by some means. In either case, it is a fatal error upon start-up.

Any activatable object that is created by a server started by hand or is created within a currently running server must register with the activator (via the activated method) before handing out its reference to any client. If this convention is not followed, the activator will not know that such an object is activated, and may start up a duplicate server with the same identity sometime in the future.

The Activator

In our protocol as defined, an activator need not maintain any persistent state, and therefore requires no recovery mechanism. There are two guarantees that the activator must make for the system to function properly:

- like all system daemons, the activator must remain running while the machine is up, and
- the activator must not start servers for those objects which the activator has been informed are active.

The activator maintains a few tables, in memory, of objects that it has participated in activating (or has been informed are activated). These tables are kept in order to enforce the latter guarantee.

The activator supports several operations: address, activate, managed, activated, isActive and deactivated. The managed operation is used in the model where there are multiple objects per server process (or shared server model). We defer discussion of managed objects to the next section.

The address operation returns the abstract location of the activator. As mentioned earlier, the representation of an address is implementation specific.

The activate and activated operations make up the core of the activation protocol. To obtain the internal object reference corresponding to some external reference, a client invokes the activate operation on the activator, located at the address represented in the token, and passes the activation token as the argument. This operation may or may not initiate

server execution depending on the status of the object (active or passive).

If the object is not already active, the activator uses the information present in token to spawn a process for the server, passing the server its identifier and activation data as arguments. If activation fails for some reason (failure to spawn server due to lack of system resources, for example), the activator raises the UnableToActivate exception to the client.

When server-side activation is complete, the server process informs the activator by invoking the activated operation on its local activator passing the object's (internal) reference as the argument. The server receives an activation identifier (of type ID_T) as a result of the operation which denotes this "instance" of activation for the object. This identifier is used in *deactivating* the object (explained below). In servicing future activation requests for the same object (with the same activation token), the activator replies with the internal reference previously reported by the server. That internal reference is valid until the object (server) deactivates.

The activator, upon receiving notification of the object's active status from the server, replies to the client with the internal reference for the active object. This internal reference is of type Activatable_T, since all activatable objects support the Activation::Activatable_T interface. The client can then narrow this reference (using a type-safe narrow), to obtain an object of the correct type on which it can invoke operations. Figure 2 illustrates the core activation protocol.

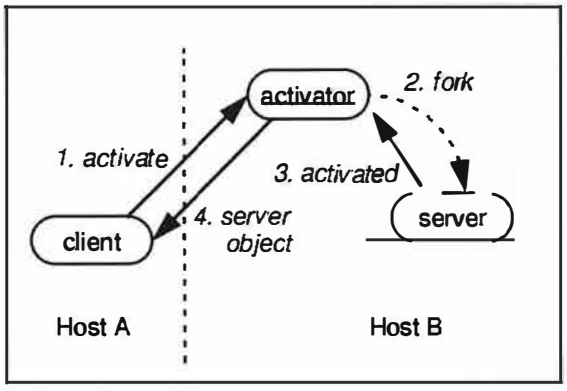


Figure 2. Core Activation Protocol

A server object that has not serviced any requests for some (lengthy) period of time, may wish to shut down or *deactivate* in order to free up system resources. In order to deactivate an object, the server calls the deactivated operation passing two parameters: the activation identifier id (returned by the activated call) and the token of the activatable object. Once an object is inactive, the activator can remove all knowledge of the object from its tables. The activation identifier returned by the activated call and supplied to the deactivated call is employed so that late or duplicate notifications of deactivation will not cause the object to be erroneously forgotten by the activator. The activator uses the identifier to distinguish between valid *instances* of the activation protocol for a particular object.

Note that it is possible for a server to crash without invoking the deactivated operation. This means that the activator will have a stale (or orphaned) internal reference for the object. Thus, clients will receive a stale object reference for the object until the activator detects the object's demise and flushes the stale reference from its tables. The network object system has a notifier facility for detecting object failure. In the activator implementation, we use this facility to detect orphaned references and clean up accordingly. If the underlying system does not provide such a facility, the activator must itself monitor server process status to clean up orphaned references.

A client can check the status of an object by invoking the isActive operation passing the activation token as an argument. This operation returns TRUE if the object corresponding to the activation token is currently executing in a process and returns FALSE if the object is currently dormant (inactive).

In each of the above mentioned activator operations, the exception WrongActivator is raised if the client directs a request to an inappropriate activator. This situation can happen due to a programmer error. Therefore, if the token parameter does not contain the same address as the one for the destination activator, then the request is not carried out and the exception is raised by that activator.

Managed Objects

Up to this point, the activation protocol described enables one active object (with the same unique identifier) per process. If the server programmer wishes to have multiple objects per process, a private protocol would have to be employed between active servers so that subsequent objects can be activated in the same process as the first one. The situation of activating a group of objects in a process is a common one, so we have enhanced the protocol, via the Manager_T interface, to handle this frequently occurring case.

A few simple additions to the basic activation protocol accomplish activation of a group of objects

in a single process (the group of objects being those that share the same executable file name). Objects using this shared server model set the isManaged flag in the activation token according to whether the object is "managed"— that is, it should be activated in the same process as other objects on the same node which share the same executable file name.

We introduce the notion of a manager of objects which is responsible for handling activation of a set of objects in a single server process. The Manager_T interface consists of one operation which embodies its activation management function.

When the activator sees an activation token indicating a managed object, it creates a process (with the switch "-exec <executableFileName>" just in case the manager can't figure this out on its own) to manage objects instead of creating one process to manage the one object. Once the manager process (executableFileName) starts up, it makes a call back to the activator, using the managed operation, passing both a reference to itself and the name of the executable for those objects which it manages.

This callback informs the activator that the manager will manage the activation of all objects with the specified executable. The activator can then forward the original activation request to the manager via the manager's activate operation. Since this call is synchronous, the manager does not need to make the activated callback to the activator; when the call returns, activation is complete and the activator can reply to the client.

Tokens subsequently received by the activator which are "managed" and have the same executable file name as a current manager will have their activation request forwarded to that manager directly via its activate operation.

The activator will still handle activate requests in the usual way if the isManaged flag is FALSE.

We could take the radical approach that all objects are managed. This approach would allow elimination of the isManaged flag in the token and the activated operation in the activator.

4 Implementation and Results

The initial implementation of the activator (a network object supporting the Activation::Activator interface) is 635 lines of Modula-3 code (including comments and excluding generic interface and module expansions). The first implementation supports concurrent activation/deactivation requests, but does not include any manager functionality (a small addition to the activator).

Performance tests were run on a dual processor SPARCstation 20 with 128 MB of memory. The first series of tests consisted of a client, the activator, and a server on the same machine.

A server object that is dormant needs to be fully activated by the activator; that is the activator needs to create a process for the server and obtain the internal reference for the object. Such *full* activation takes 700 milliseconds for client, activator and server running on the same machine, all in separate address spaces. This timing includes round-trip latency between client and activator, and process creation time. Most of the overhead associated with activation is spawning a new process for the object server which takes roughly 550 milliseconds. Additionally, activation time is effected by the amount of time the server takes to initialize itself before replying to the activator that the server has completed activation (by invoking the operation activated). We keep the server initialization time to a minimum, only performing the following steps in the server: parse arguments, import the activator, create a server object, and reply immediately to the activator. Such server-side initialization takes about 100 milliseconds. Table 1 shows the breakdown for full activation on a single machine.

If an object server is currently running (that is previously activated via the activator), the activation procedure consists of the activator handing back to the client an in-memory reference for the server object. For client, activator, and object server on the same machine, this scenario takes 3.6 milliseconds and mostly reflects the time for a remote method invocation between client and activator.

When the client and activator are on different machines, full activation takes 692 milliseconds. For this same configuration, handing to the client an already activated object takes 4.6 milliseconds. It is interesting to note that full activation for a remote server takes less time than activation where both client and server reside on a single machine. The increased latency for the local case is due to contention for system resources between client and activator. Table 1 (below) summarizes our performance results (in milliseconds).

Activation performance:	
full activation (local)	700
full activation (remote)	692
simple activation (local)	3.6
simple activation (remote)	4.6

Breakdown for full activation:	
fork server	550
server execution	100
other activation overhead (including: GC ~4 ms; and communication with client ~3-5 ms)	40
Other baseline measurements:	
fork trivial Modula-3 program	151
fork empty Perl script	71
fork trivial C++ program	61
fork empty Tcl script	52
fork trivial C program	24

Table 1. Activation Results (in milliseconds)

5 Related Work

The Common Object Request Broker Architecture (CORBA [3]) outlines a protocol for activation. This protocol, specified as part of the Basic Object Adaptor (BOA), attempts to be a completely general solution for activation providing sophisticated activation dependency features such as co-activation and group activation. While these features may be needed in some applications, our experience has shown that a simpler protocol suffices for most applications; we have chosen to focus on a clean protocol rather than an ideal API.

The CORBA activation protocol embodies a multitude of activation features which puts an unnecessary burden upon both server implementations and implementations of the activation mechanism itself, if such extended features are not required by server applications. We use a simpler model that supports activation of a single object within a server, or multiple "managed" objects within a server process. If more complex functionality is required, this functionality can be layered upon our activation protocol.

6 Discussion

Object mobility. Currently, an activation token for an object can not change over time (tokens are fixed when created). Since the location of the server machine is embedded in the token (implicitly in the activator address), objects can not be moved to another machine once a token for that object has been handed out. To overcome the restriction on mobility, a level of indirection could be built into the token, but this could have a significant impact on the performance and reliability of the activation process.

As an alternative form of indirect reference, names could be used as external references for long-lived objects. Thus given a name, a client can contact a name service to obtain an internal reference to an active object currently associated with that name.

Another scheme that supports object mobility is one in which a relocated object leaves, at its old location, a *forwarding pointer* (or "tombstone") [5] that indicates the new location for an object. Using forwarding information left for an object, an activator receiving activation requests at an object's old location can redirect such requests to its new location for a period of time.

The problems with this approach are three-fold. First, the indirection introduced by employing forwarding pointers can increase the overhead of activation, since each activator must deal with both servicing and forwarding requests; this dual-role can cause a potential bottleneck at the activator. Additionally, multiple redirections may occur for requests involving frequently relocated objects, thus increasing the overall service time for activation requests. Finally, forwarding information must eventually be removed, but determining when it is "safe" to remove such information is difficult; there is no easy way to determine how many objects, and specifically which objects, contain references to a relocated object's previous address.

Server implementation requirements. Activation is not a passive protocol. It requires participation and cooperation on the part of the server. Ill-behaved servers can potentially cause deadlock during activation of that server. Servers that fail to inform the activator when they become active can cause unpredictable occurrences such as duplicate servers executing with the same identity. Since we allow servers to be started without activator assistance, informing the activator of an object's status becomes even more vital.

Type system. In our system, external references can be exposed to both clients and servers alike. This additional form of reference is outside the type system and therefore can not be checked statically. In large scale systems, non-type-checked references pose many implementation hazards. If external references were completely hidden from the programmer, this would not be as much of an issue. Since no real type information is present in a token, the client and server programs must be careful to perform a typesafe narrow on a reference resulting from an activation request. An activation strategy that utilized activation tokens that are typed to reflect the objects they represent would overcome this deficiency. Thus ideally, either external references need to be

integrated with the type system or the runtime system must hide this form of reference from the programmer.

Need more help from the runtime. The lesson here is that you really do need help from the runtime to make activation seamless. Lazy activation is preferable, and is not easily layered on an already existing system (such as network objects).

As an optimization, the runtime could transmit along with the object its activation token, so that it is more easily accessible to a remote process, and a remote method invocation is not required to obtain the token. This saves times as well as eliminates a potential failure mode if the object from which the token is being obtained could not be reached for some reason (e.g., network failure or processor crash). This scheme would require modifications to the runtime to be completely transparent to the client. Of course, the activation token could be transmitted as a parameter in all interfaces, but this exposes activation at the wrong level.

The problem with making the decision not to tweak the runtime is that artifacts of our activation protocol end up creeping into other interfaces that we design (as with UUIDs needing to be transmitted along with a network object references). If we modified the runtime, object identifiers could be transmitted along with object references. If these two were transmitted as one unit, then testing for equality would be little overhead and not require a remote method invocation to obtain the identifier for comparison. Also, we would not have to expose object identifiers explicitly in our interfaces since objects would inherently have a determined, easily accessible identity.

7 Availability

The Modula-3 source code for the activator and other libraries is freely available. In order to obtain the release, please contact the authors for more information.

References

[1] Birrell, Andrew, Greg Nelson, Susan Owicki, and Edward Wobber, "Network Objects," Digital Equipment Corporation Systems Research Center Technical Report 115 (1994).

[2] Nelson, Greg (ed.), *Systems Programming with Modula-3*, Prentice Hall (1991).

[3] The Object Management Group. "Common Object Request Broker: Architecture and Specification." *OMG Document Number 91.12.1* (1991).

[4] Hosking, Antony L., and J. Eliot B. Moss, "Towards Compile-Time Optimisations for Persistence." In *Implementing Persistent Object Bases: Principles and Practice—The Fourth Int'l Workshop in Persistent Object Systems*, Morgan Kaufmann Publishers, Inc. (1990), pp. 17-27.

[5] Jul, Eric, Henry Levy, Norman Hutchinson, and Andrew Black, "Fine-Grained Mobility in the Emerald System." In *ACM Transactions on Computer Systems 6*, 1 (February 1988), pp. 109-133.

Dynamic Insertion of Object Services

Ajay Mohindra — *IBM T. J. Watson Research Center, Hawthorne, NY, ajay@watson.ibm.com*

George Copeland — *IBM Austin, 11400 Burnet Road, Austin, TX, copeland@austin.ibm.com*

Murthy Devarakonda — *IBM T. J. Watson Research Center, Hawthorne, NY, mdev@watson.ibm.com*

Abstract

This paper presents our experiments with *dynamic insertion* of object services, where dynamic insertion is defined as adding services to object instances at runtime. In contrast, the *static* approach is defined as adding services to objects at *class definition* time. Dynamic insertion allows class implementors to concentrate on the basic functionality for objects, freeing them from the chores of providing normal system services such as persistence, transactions, and concurrency. This paper compares the dynamic insertion approach to the static approach using two benchmarks. The two key contributions of the paper are: (1) It shows how to use the dynamic insertion of services in well known benchmarks; (2) It demonstrates that the dynamic insertion approach incurs low overhead.

1 Introduction

In the Common Object Service Specification document [1] and the Object Services Architecture document [2], Object Management Group (OMG) has specified several object services including persistence, concurrency, transactions, and security. Class implementors can use these services in developing new classes to meet their requirements. At the present, implementors incorporate necessary services to application objects at the class definition time. One disadvantage of this approach is that several classes have to be defined to provide different combinations of services. For example, for a base class called *PhoneDirectory*, a class implementor would have to write a new class called *PersistentPhoneDirectory* to support persistence (inheriting from *PhoneDirectory* and *PersistentObject* classes), a class called *TransactionPhoneDirectory* to support transactions (inheriting from *PhoneDirectory* and *TransactionObject* classes), and a class called *TransactionPersistentPhoneDirectory* to support both transactions and persistence (inheriting from *PhoneDirectory*, *PersistentObject* and *TransactionObject* classes). As can be seen, for every new service, all possible combinations of services doubles the number of classes that need to be written. A total of $2^{\mathcal{P}}$ classes are needed to provide all combinations of $\mathcal{P}$ services. For $\mathcal{N}$ different base classes, this results in a total of $\mathcal{N} * 2^{\mathcal{P}}$ classes–an unnecessary increase in complexity and size of the class implementor's class diagram and library.

Copeland [3] argues that a class implementor should be only concerned with writing code for the base class without worrying about system services such as persistence, concurrency, and transactions. At the time of object instantiation, a user of the class should be able to specify needed services for an object instance. The object runtime would then

add requested services to the object instance. This approach of adding services to objects at the instantiation time is called *dynamic insertion* of object services. In contrast, incorporating services at the class definition time is referred to as the *static* approach in this paper. The main advantages of the dynamic insertion approach are:

- It reduces the number of classes to be written by a class implementor to support all combinations of services–For supporting $\mathcal{P}$ services, only $\mathcal{N} + \mathcal{P}$ classes are needed as compared to $\mathcal{N} * 2^{\mathcal{P}}$ classes in the static approach;
- It allows for rapid development of object systems as class implementors focus on writing code to provide only the basic functionality of the objects; and
- It improves programmer productivity by encouraging reuse of system provided services.

Being a novel approach, many people are not aware of how to use the dynamic approach for building object systems. In this paper, we present our experience in using the dynamic insertion of object services for two standard benchmarks. The two benchmarks are implemented in IBM's System Object Model (SOM) [4]. We use OMG's persistence service [1] as an example. We provide performance comparisons of the static and dynamic insertion approachs, and discuss the conditions under which one is better than the other.

The rest of the paper is organized in the following manner. Section 2 briefly describes implementation of the dynamic insertion approach in SOM. Section 3 describes the benchmarks used in the study. Section 4 presents performance measurements and Section 5 presents discussion of the results. Conclusions and future work are presented in Section 6.

2 Dynamic Insertion of Services

In the dynamic insertion approach, a user specifies the set of services to be added to an object instance. The set of services are specified as arguments to the `create_instance()` call. The system runtime, in turn, generates a new hybrid class with the set of services added and returns an instance of the new class. In SOM, dynamic insertion can be implemented using the following two runtime mechanisms:

- Dynamic Subclassing – A new class is dynamically created (at the runtime) by mixing the original class and a special class that provides the required service.
- BeforeAfter Metaclass – The BeforeAfter metaclass allows a *before* method and an *after* method to be inserted around each method of the original class. The before and after methods can be customized to do any service specific tasks to implement the desired functionality (such as locking and unlocking for providing concurrency). Further, multiple BeforeAfter metaclasses can be used to implement a combination of services. For details on the BeforeAfter metaclass mechanism in SOM, the reader is referred to [5].

An example will help illustrate the two mechanisms. Consider a *Directory* class that had been initially written by a class provider with no persistence in mind. The *Directory* class supports three operations: `lookup`, `insert`, and `remove`. A user wants to use the *Directory* class for an application but additionally wants the directory to be persistent. One approach the user can take is to rewrite the *Directory* class by subclassing it from a base class that supports persistence (see Figure 1a). The base persistent class provides necessary interfaces for saving and restoring the *Directory* class. The disadvantage of this hand-crafted approach is that any new combination of services would require the user to rewrite the *Directory* class taking into account all the needed services. This unnecessarily complicates and increases the size of the class diagram–every new service causes the number of classes to double.

Under SOM, similar effect can be achieved at runtime using the subclassing mechanism. Using the *Directory* class as an example, if the user wants to make the *Directory* object instance persistent then

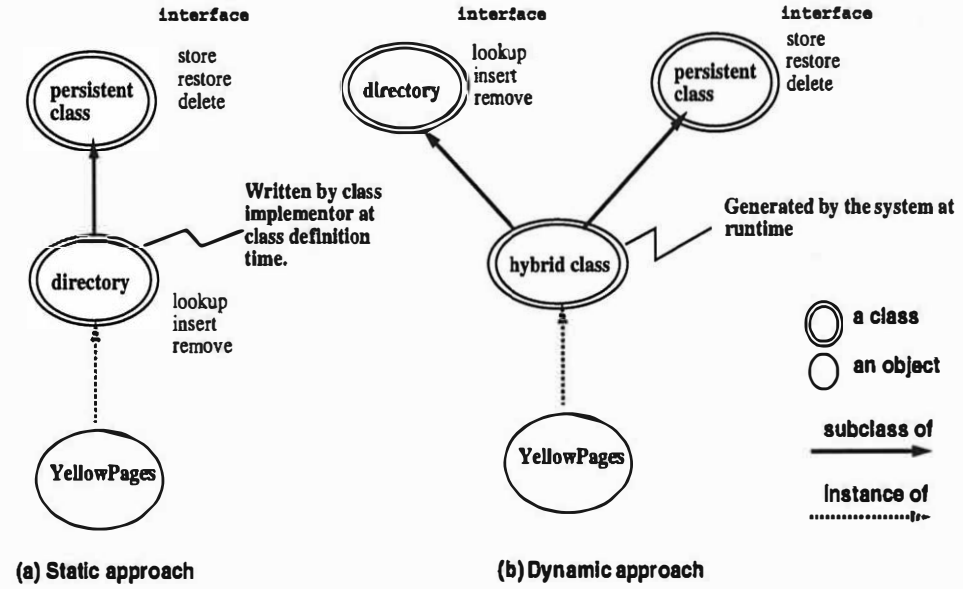


Figure 1: Two approaches for creating a persistent directory object.

the user would specify the need for the persistence service at the time of instantiating the object. The SOM runtime would then generate a new hybrid class from the *Directory* class by mixing in a special class that provides the persistence service (see Figure 1b). This approach automatically inserts the required set of services at runtime, thereby, reducing the amount of work that needs to be done by the user. Figure 2 shows the pseudo-code for the SOM runtime.

The SOM BeforeAfter metaclass mechanism can also be used for dynamically inserting services although the mechanics are slightly differently from the subclassing mechanism. The BeforeAfter metaclass allows the insertion of a *before* method and an *after* method around each method of the original class. This facilitates adding services that have a well-defined "before" and "after" effect. For example, if an object wants to implement concurrency control for all its methods then the *before* method can ensure that the object is properly locked prior to the execution of the method; and the *after* method can do the unlock subsequent to the completion of the method. A variety of such BeforeAfter metaclasses can be written *a priori* and used for adding the desired set of services to the objects.

In contrast to the two dynamic mechanisms described above, the static approach requires the services to be inserted at class definition time. The next section describes the two benchmarks used in performance comparison of the static and dynamic approaches.

3 Benchmarks

We used the OO1 benchmark [6] and TPCB benchmark [7] for the evaluation. They were chosen to measure the effectiveness of the two approaches but not to measure the performance of the underlying database subsystem.

3.1 OO1 benchmark

OO1 was the first benchmark to measure the performance of engineering databases. The benchmark operates on a database consisting of part objects and connections between them. Each part has five fields: a part-id, a part type, an (x, y) coordinate pair, and a build date. The connections capture the linkage between different parts. Each part has three outgoing connections to other parts, and a variable number of incoming connections from other parts. Each connection is represented by a connection type and a length. To provide a notion of locality, all

Application Program

```
SOMObject *obj;
/* Specify the class name and
list of services to add */
obj = create_instance(ClassName, svce1, svce2, ...);

method1(obj, ...);        /* Invoke methods */
```

SOM Runtime

```
Lookup internal directory to see if a class
with name ClassName exists with services
svce1 and svce2 added

If such a class exists
    then create and return an instance
else {
    Create a new class which inherits from
    ClassName and other classes that
    provide services svce1 and svce2.

    Initialize and Register the new class.

    Return an instance of the new class.

    Add the new class to the internal
    directory for future use.
}
```

Figure 2: Pseudo-code for the dynamic insertion approach.

parts are logically ordered by part-id, and 90% of the outgoing connections are to 1% of parts with part-id values "closest", while the remaining 10% of connections are made to any randomly selected part.

The benchmark consists of three tasks. The first task randomly selects 100 parts, retrieves each part and performs an operation on the part. The second task performs forward traversal--one part is selected at random and all connections from it to other parts are traversed in a depth-first manner up to seven levels. An operation is performed on each part encountered during this traversal. A similar traversal is also performed in the reverse order, starting with a randomly selected part and retrieving all the parts connected to it, and so on. In the third task, 100 new parts are added to the database with 3 connections from each part to other parts, and then updates are committed.

The results were recorded for a "cold start", where the database existed only on disk, and a "warm start", where measurements are taken after ten iterations of each task. For our measurements, we used a database with 20,000 parts and 60,000 connections.

SOM implementation

In the SOM implementation, the data was stored in the IBM's UNIX relational database sys-

```
interface Part : PersistentObject,
                 StreamableObject {
 attribute long      id;
 attribute string    type;
 attribute long      x;
 attribute long      y;
 attribute long      build;
 void set_values(in long id, in char *type,
                 in long x, in long y,
                 in long tm);
 void connect( in Part p, in Connection c);
 void print_part();
};
interface Connection : PersistentObject,
                       StreamableObject {
 attribute string    type;
 attribute long      id;
 attribute long      length;
 attribute long      source;
 attribute long      target;
 void set_values(in long index, in long length,
                 in char *type);
 void assign(in Connection c);
 void print_connection();
};
```

Figure 3: IDL for the Part and Connection classes.

tem, DB2/6000 [8]. The part and connection were written as two separate classes with the individual fields as class attributes. In the static approach, the part and connection classes were subclassed from a *PersistentObject* class, while in the dynamic approach the part and connection classes were subclassed from the base *SOMObject* class. Figure 3 shows the IDL for the static approach. The *StreamableObject* class shown in the IDL is to ensure that OMG's Externalization service is available for the part and connection classes. The *StreamableObject* class provides interfaces to get data in and data out of an object. For the dynamic insertion approach, we used the same IDL except the two classes did not inherit from the *PersistentObject* class. The persistent service, however, was added to the object at the time of instantiation using the dynamic subclassing mechanism.

3.2 TPCB benchmark

The TPCB benchmark from the Transaction Processing Performance Council measures the OLTP system performance. It is based on the TPCA benchmark with the exception that no terminals and networking components are included. TPCB is designed only to exercise the basic components of a database system. TPCB represents transactions on the database belonging to a hypothetical bank that has one or more branches. Each branch has multiple tellers. The bank has many customers each with an account. The database maintains the cash position of the branch, teller, and account, and a history of recent transactions at the bank. The transaction represents the work done when a customer makes a deposit or a withdrawal against his or her account. Figure 4 shows the algorithm for the TPCB benchmark. For our measurements, we used a database with 1 branch, 10 tellers, and 100,000 accounts.

```
Randomly select an account Aid, a teller Tid,
a branch Bid and Delta.
BEGIN TRANSACTION
  Update Account where AccountID = Aid:
    Read AccountBalance from Account
    Set AccountBalance = AccountBalance + Delta
    Write AccountBalance to Account
  Write to History:
    Aid, Tid, Bid, Delta, TimeStamp
  Update Teller where TellerID = Tid:
    Set TellerBalance = TellerBalance + Delta
    Write TellerBalance to Teller
  Update Branch where BranchID = Bid:
    Set BranchBalance = BranchBalance + Delta
    Write BranchBalance to Branch
COMMIT TRANSACTION
```

Figure 4: Algorithm for the TPCB benchmark

SOM implementation

In the SOM implementation, the data was stored as four tables in IBM's DB2/6000. The account, history, teller, and branch were written as separate classes. Similar to the OO1 implementation, in the static approach, these four classes inher-

Table 1: Measurements for creating and deleting an instance of the *part* object for object sizes 36 bytes and 32K bytes (The times are given in microseconds).

Obj Size	Ops.	Static	Dynamic	
			Naive	File-Away
36 bytes	Create	106	2630	146
	Destroy	32	38	38
32K bytes	Create	1566	4235	1610
	Destroy	394	415	409

ited from the *PersistentObject* class while in the dynamic approach, persistence was added at the time of object instantiation using the dynamic subclassing mechanism. Figure 5 shows the IDL for the four classes for the static approach. Locking was implemented using the AIX operating system semaphores. The *StreamableObject* class shown in the IDL is to ensure that OMG's Externalization service is available for the objects.

4 Measurements

The measurements reported in this section were taken on an IBM RISCSystem/6000 with 256M bytes RAM. For the persistence storage, we used IBM's relational database, DB2/6000. The benchmarks were implemented in SOM version 2.1GA. We divided our measurements into two categories: the first category measures the cost of creating and deleting an object instance, while the second category measures the performance of the benchmark.

Table 1 shows the timings for creating and destroying an instance of an object using the two approaches. For the dynamic approach, we took two sets of measurements. Times under column labeled "naive" are for the strategy in which each `create_instance` operation results in a new hybrid class (see Figure 1) being generated, i.e. the system performs all the necessary operations for creating an instance. Column labeled "file-away" corresponds to

```
interface Account : PersistentObject,
                    StreamableObject {
  attribute long      accountid;
  attribute long      balance;
  attribute long      branchid;
  void update_balance(in long delta);
};
interface Branch : PersistentObject,
                   StreamableObject {
  attribute   long    branchid;
  attribute   long    balance;
  void set_values(in long branchid,
                  in long balance);
  void update_balance(in long delta);
};
interface History : PersistentObject,
                    StreamableObject {
  attribute       long    accountid;
  attribute       long    tellerid;
  attribute       long    branchid;
  attribute       long    delta;
  attribute       long    time;
 void set_values(in long accountid,
                 in long tellerid,
                 in long branchid,
                 in long delta,
                 in long time);
};
interface Teller : PersistentObject,
                   StreamableObject {
 attribute   long    tellerid;
 attribute   long    balance;
 attribute   long    branchid;
 void set_values(in long tellerid,
                 in long balance,
                 in long branchid);
 void update_balance(in long delta);
};
```

Figure 5: IDL for the Account, Teller, History and Branch classes.

Table 2: Performance of the OO1 benchmark.

Time in seconds				
Operation		Static	Dynamic	
			Naive	File-Away
lookup	Cold	7.0	9.6	6.8
	Warm	2.2	4.1	2.2
forward	Cold	54.0	85.0	60.0
traversal	Warm	44.0	59.0	45.0
reverse	Cold	49.0	102.0	62.0
traversal	Warm	39.0	79.0	40.0
insert	Cold	4.8	6.4	4.6
	Warm	3.4	4.2	3.4

Table 3: Performance of the TPCB benchmark.

Units are transactions per second		
Static	Dynamic	
	Naive	File-Away
13.73	13.91	13.91

the strategy in which the system does not treat each `create_instance` operation as a new one, but instead first looks through an internal directory to see if the hybrid class with the specified set of services had been created before. If such a class exists then that class is used for creating a new instance. If such a class does not exist then the system creates a new hybrid class and files it away for future use. As one can see, the file-away strategy optimizes the `create_instance` operation by reducing the amount of work in subsequent instantiations.

Indeed, Table 1 shows that it is more expensive to create or destroy an object instance using the dynamic approach. The additional cost is especially high when using the naive approach, and the additional cost is relatively small using the file-away strategy. The additional costs are due to the operations that are need to create the new hybrid class. These operations include generating a new metaclass, defining the new class data structure, and creating a new instance. The file-away strategy reduces most of the associated cost. When the object is large, the overhead is quite small as Table 1 shows (only about 3.5% for a 32K byte object). For the static approach, most of the work needed to create a new class is performed at the compile time. The performance characteristics of the destroy operation are similar.

Table 2 shows the performance of the three tasks for the OO1 benchmark. The lookup task involves looking up 1000 randomly chosen parts in the parts database. The forward and reverse traversals perform traversal seven levels deep into the parts database, and the insert task inserts 100 new parts into the parts database. Table 2 shows that the performance of the static approach is consistently better than the dynamic approach when compared to times for the naive strategy. When using the file-away strategy, however, the performance for the lookup and insert tasks in the static and dynamic approach is approximately the same. The performance is still poor in the dynamic approach for the traversal tasks for "cold start". The result was surprising at first because the only difference in the two approaches is that the dynamic approach has a higher cost for creating object instances. Further investigation revealed that the overhead associated with creating object instances during traversal was dominating the overall time for a "cold start". During the traversal tasks, more object instances were created than during the lookup and insert tasks. Once the memory was populated with newly created objects, the performance of the dynamic approach for the "warm start" was similar to the static approach.

Table 3 shows the performance of the TPCB benchmark. The benchmark was run on an account database consisting of 100,000 accounts using multiprogramming level of one. The benchmark creates one instance each of the account, branch, teller and history objects. These four instances are used to run all the transactions in the benchmark. Thus, the difference in creation costs contributes a negligible portion to the performance of the benchmark. As a result, the two approaches show similar per-

formance. The main overhead for the TPCB benchmark lies in the heavyweight database operations that are performed.

5 Discussion

In the previous section, we presented performance measurements of the static and dynamic approaches using two standard benchmarks. The results indicate that the dynamic approach can potentially incur significant overhead for creating object classes on the fly. However, the overhead associated with the dynamic approach can be reduced to a small percentage by using a file-away strategy where newly created hybrid classes are reused internally. The file-away strategy, however, introduces overheads of its own: First, the storage overhead for storing the hybrid classes, and second, search overhead for searching through a set of hybrid classes. These overheads can be minimized by using LRU-based garbage collection, and efficient search strategies. For applications that create few new object instances or for applications that create large objects, the performance difference between the static and the dynamic approach is quite small. Thus, dynamic insertion of object services offers a promising approach for designing large object systems. Of course, the main advantage of the dynamic approach is that it reduces the number of classes that a programmer has to write; allows class implementors to focus on the core functionality of the objects; and increases programmer productivity by encouraging reuse of system provided services. The dynamic approach can, thus, be used for rapid prototyping of complex applications.

6 Summary

In this paper, we presented a dynamic approach to adding services to objects, showed how to use the approach for two standard benchmark programs, and demonstrated that the dynamic approach can be implemented while incurring only a small overhead. The main advantage of the dynamic approach is that it allows the class implementors to concentrate the basic functionality of the objects, freeing them from the chores of incorporating system services. The dynamic approach also reduces the number of classes a class implementor has to write to provide different combinations of $\mathcal{P}$ services for $\mathcal{N}$ different classes (a total of $(\mathcal{N}+\mathcal{P})$ classes for the dynamic approach as compared to $(\mathcal{N} * 2^{\mathcal{P}})$ classes for the static approach). The two approaches have been compared using the OO1 and TPCB benchmarks. The results demonstrate that a file-away strategy for the dynamic insertion incurs only a small additional performance penalty over the static approach. The key advantage of the dynamic approach is that it allows for rapid prototyping of object systems and improves programmer productivity. As part of the future work, we are investigating the performance of the BeforeAfter metaclass mechanism, and the effects on the dynamic approach when two or more services are simultaneously requested.

References

[1] Object Management Group. Common Object Services Specification Volume 1. Technical Report 94-1-1, Mar. 1994.

[2] Object Management Group. Object Services Architecture. Technical Report 94-11-12, Dec. 1994.

[3] George Copeland. System services for SOM objects. Technical Report, IBM Austin, Oct. 1994.

[4] IBM, 11400 Burnet Road, Autsin, TX. *SOMObjects Developer Toolkit Users Guide*, Oct. 1994.

[5] I.R. Forman, S. Danforth, and H. Madduri. Composition of before/after metaclass in SOM. In *Object Oriented Programming Systems, Languages, and Applications*, Oct. 1994.

[6] R.G.G. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database systems*, 17(1):1–31, Mar. 1992.

[7] Jim Gray. *The benchmark handbook: for database and transaction processing systems.* Morgan Kaufman, 1991.

[8] IBM. *Database 2 AIX/6000 Programming Guide*, Oct. 1993.

Object-Oriented Components for High-speed Network Programming

Douglas C. Schmidt, Tim Harrison, and Ehab Al-Shaer
schmidt@cs.wustl.edu, harrison@cs.wustl.edu, and ehab@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130
(314) 935-7538

Abstract

This paper makes two contributions to the development and evaluation of object-oriented communication software. First, it reports benchmark results from several network programming mechanisms (such as sockets and CORBA) on Ethernet and ATM networks. These results illustrate that developers of bandwidth-intensive and delay-sensitive applications (such as interactive medical imaging or teleconferencing) must evaluate their performance requirements and the efficiency of their communication infrastructure carefully before adopting a distributed object solution. Second, the paper describes the software architecture and design principles of the ACE object-oriented network programming components. These components encapsulate UNIX and Windows NT network programming interfaces (such as sockets, TLI, and named pipes) with C++ wrappers. Developers of object-oriented communication software have traditionally had to choose between high-performance, lower-level interfaces provided by sockets or TLI or less efficient, higher-level interfaces provided by communication frameworks like CORBA or DCE. ACE represents a midpoint in the solution space by improving the correctness, programming simplicity, portability, and reusability of performance-sensitive communication software.

1 Introduction

Distributed object computing (DOC) frameworks like the Common Object Request Broker Architecture (CORBA) [1], OODCE [2], and OLE/COM [3] are well-suited for applications that exchange richly typed data via request-response or oneway communication. However, current implementations of DOC frameworks may be less suitable for an important class of bandwidth-intensive and delay-sensitive applications that stream relatively simple datatypes over high-speed networks. Medical imaging, interactive teleconferencing, and video-on-demand are common examples of these streaming applications.

Streaming applications with stringent throughput and delay requirements are ideal candidates for high-speed networks such as ATM and FDDI. However, these applications may not be able to tolerate the overhead associated with contemporary DOC frameworks. This overhead stems from non-optimized presentation layer conversions, data copying, and memory management, inefficient receiver-side demultiplexing and dispatching operations, synchronous stop-and-wait flow control, and non-adaptive retransmission timer schemes. Meeting the throughput demands of streaming applications has traditionally involved direct access to network programming interfaces such as sockets [4] or System V TLI [5]. These lower-level interfaces are efficient since they omit unnecessary functionality (such as presentation layer conversions for ASCII data) and allow fine-grained control over memory management, protocol buffering, demultiplexing, and flow control.

However, conventional network programming interfaces are low-level, non-portable, and non-typesafe, which complicates programming and permits subtle run-time errors. For instance, communication endpoints in the socket interface are identified by weakly-typed integer *handles* (also known as *socket descriptors*). Weak type-checking increases the potential for run-time errors since compilers cannot detect or prevent improper use of handles. Thus, operations can be applied to handles incorrectly (such as invoking a `read` or `write` on a passive-mode handle that can only accept connections).

Traditionally, developers of high-performance streaming applications had to choose between two solutions:

1. *Higher-level, but less efficient network programming interfaces* – such as DOC frameworks or RPC toolkits;
2. *Lower-level, but more efficient network programming interfaces* – such as sockets or TLI.

This paper describes object-oriented network programming components that provide a midpoint in the solution space. These components are part of the ACE toolkit [6], which encapsulates conventional network programming interfaces with a family of C++ wrappers. The ACE toolkit improves the correctness, ease of use, portability and reusability of communication software without sacrificing performance.

This paper is organized as follows: Section 2 compares the performance of several network programming mechanisms (C sockets, C++ wrappers for sockets, and two implementations of CORBA) for a representative streaming application

over Ethernet and ATM networks; Section 3 outlines the design of the object-oriented ACE components that encapsulate UNIX and Windows NT network programming interfaces (such as sockets, TLI, STREAM pipes, and named pipes); Section 4 illustrates the differences between programming with C sockets, ACE, and CORBA; Section 5 summarizes the design principles of the ACE wrappers; and Section 6 presents concluding remarks.

2 Performance Experiments

This section describes performance results from comparing several network programming mechanisms that transfer large streams of data using TCP/IP over Ethernet and ATM networks. The network programming mechanisms compared below include C sockets, C++ wrappers for sockets, and two implementations of CORBA. The benchmark tests are representative of applications written by the authors for the Motorola Iridium project [7] (which is a next-generation satellite-based global personal communication system) and Project Spectrum (which is an enterprise-wide medical imaging system that transports radiology images across high-speed ATM LANs and WANs [8]).

2.1 Test Platform and Benchmarks

The performance results in this section were collected using a Bay Networks LattisCell 10114 ATM switch connected to two uni-processor SPARCstation 20 Model 50s. This LattisCell is a 16 Port, OC3 155Mbs/port switch. The SPARCstations contain 100 MIP Super SPARC CPUs running SunOS 5.4. The SunOS 5.4 TCP/IP protocol stack is implemented using the STREAMS communication framework [9]. Each SPARCstation 20 has 64 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Mbits/sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) size of a SONET frame on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. 32 Kbytes is alloted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64K). This allows up to 8 connections per card.

Data for the experiments was produced and consumed by an extended version of the widely available `ttcp` [10] protocol benchmarking tool. This tool measures end-to-end data transfer throughput in Mbps from a transmitter process to a remote receiver process. The flow of user data is unidirectional, with the transmitter flooding the receiver with a user-specified number of data buffers. Various sender and receiver parameters (such as the number of data buffers transmitted, the size of data buffers, and the size of the socket transmit and receive queues) may be selected at run-time.

The following versions of `ttcp` were implemented and benchmarked:

- *C version* – this is the standard `ttcp` program implemented in C. It uses C socket calls to transfer and receive data via TCP/IP.

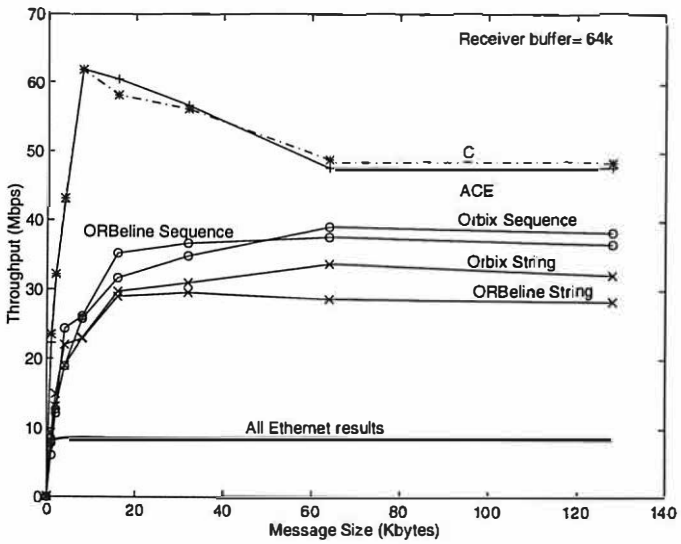


Figure 1: C, ACE, Orbix and ORBeline Performance over ATM and Ethernet

- *ACE version* – this version replaces all C socket calls in `ttcp` with the C++ wrappers for sockets provided by the ACE network programming components (version 3.2) [6]. The ACE wrappers encapsulate sockets with efficient and typesafe C++ interfaces.
- *CORBA versions* – two implementations of CORBA were used: version 1.3 of Orbix from IONA Technologies and version 1.2 of ORBeline from Post Modern Computing. These versions replace all C socket calls in `ttcp` with stubs and skeletons generated from a pair of CORBA IDL definitions. One IDL definition uses a `sequence` parameter for the data buffer and the other uses a `string` parameter.

Each version of `ttcp` was compiled using SunC++ 4.0.1 with the highest level of optimization (`-O4`). To control for confounding factors the timing mechanisms, command-line options, socket options, and communication protocols were held constant for all implementations of `ttcp`. Only the connection establishment and data transfer mechanisms were varied.

2.2 Results

We ran a series of tests that transferred 64 Mbytes of user data in buffers ranging from 1 byte to 128 Kbytes using TCP/IP over Ethernet and ATM networks. Data buffers were run in increments of 1 byte, 1K, 2K, 4K, 8K, 16K, 32K, 64K, and 128K sizes. Two different sizes for socket queues were also used: 8K (the default on SunOS 5.4) and 64K (the maximum size supported by SunOS 5.4). Each test was run 20 times to account for performance variation due to transient load on the networks and hosts. The variance between runs was very low since the tests were conducted on otherwise unused networks.

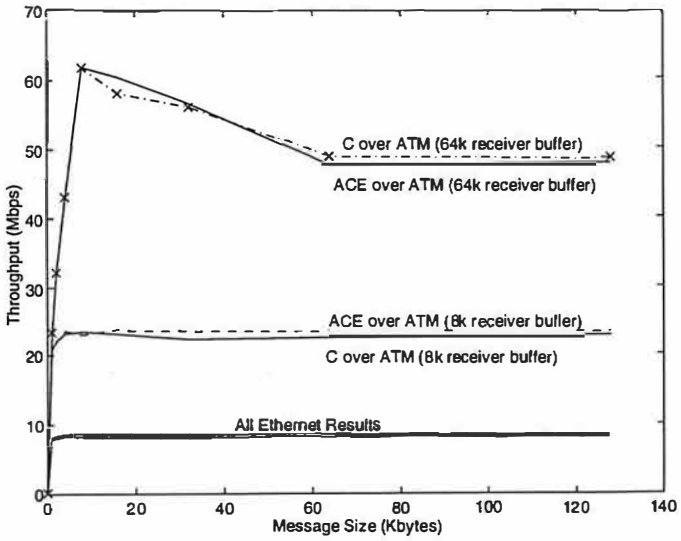


Figure 2: C and ACE Performance over ATM and Ethernet

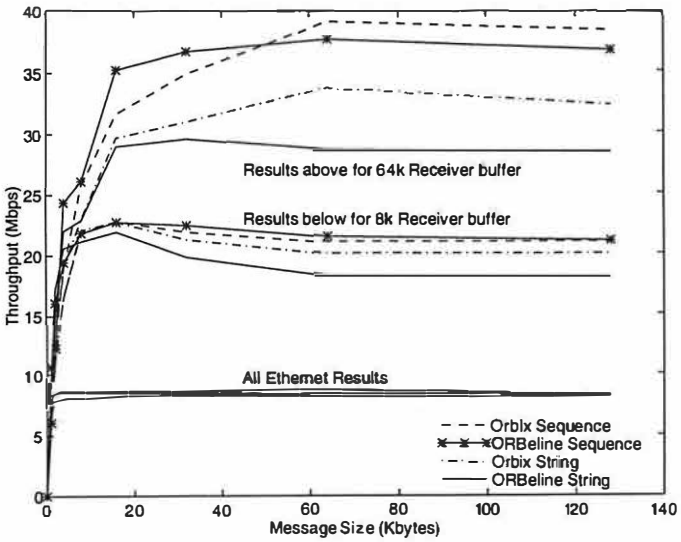


Figure 3: Orbix and ORBeline Performance over ATM and Ethernet

Figure 1 summarizes the performance results for all the benchmarks using 64K socket queues over a 155 Mbps ATM link and a 10 Mbps Ethernet (the 8K socket queue results are presented below and Tables 1 and 2 summarize the results for all the tests). The C and ACE C++ wrapper versions of `ttcp` obtained the highest throughput: 62 Mbps using 8K data buffers. In contrast, the Orbix and ORBeline CORBA versions of `ttcp` peaked at around 39 Mbps with 64K data buffers using IDL `sequences`.

The results for Ethernet show much less variation, with the performance for all tests ranging from around 8 to 8.7 Mbps with 64K socket queues. None of the Ethernet benchmarks ran faster than 8.7 Mbps, which is 87 percent of the maximum speed of a 10 Mbps Ethernet. Although the absolute throughput of `ttcp` is much faster over ATM, the relative utilization of the network channel speed was much lower (62 Mbps represents only 40 percent of the 155 Mbps ATM link).

The disparity between network channel speed and end-to-end application throughput is known as the *throughput preservation problem* [11], where only a portion of the available bandwidth is actually delivered to applications. This problem stems from operating system and protocol processing overhead (such as data movement, context switching, and synchronization [12]). As shown in Section 2.2.2, the throughput preservation problem is exacerbated by contemporary implementations of DOC frameworks like CORBA, which copy data multiple times during fragmentation/reassembly, marshalling, and demarshalling.

Sections 2.2.1 and 2.2.2 examine these performance results in detail and Section 2.3 presents recommendations based on the results.

2.2.1 C and ACE Wrapper Implementations of TTCP

Figure 2 illustrates the performance results from the C and ACE wrapper versions of `ttcp` over ATM and Ethernet. The performance of C sockets and ACE C++ wrappers are roughly equivalent, indicating there is no significant performance penalty for using the ACE wrappers. Both peak at 62 Mbps over ATM using 8K data buffers and 64K socket queues. When the data buffers exceeded 8K performance began to decline, leveling off at around 48 Mbps with 64K data buffers. This behavior is caused primarily by the MTU size of the ATM network, which is 9,180 bytes (the MTU size of a SONET frame). When data buffers exceed the MTU size they are fragmented and reassembled, thereby lowering performance.

Figure 2 also illustrates the impact of socket queue size on throughput. Larger socket queues increase the TCP window size, which allows the transmission of multiple TCP segments back-to-back. In the case of ATM, increasing the socket queue from 8K to 64K improves `ttcp` performance significantly from 23 Mbps to 62 Mbps.

The Ethernet results for large and small socket queues are more similar than the ATM results. They peak at 8.4 Mbps with 8K socket queues and 8.7 Mbps with 64K socket queues. In both cases, the factor limiting performance is the slow speed of the network.

2.2.2 CORBA Implementations of TTCP

Figure 3 illustrates the results of measuring two versions of `ttcp` implemented with two different versions of CORBA. The CORBA implementations were developed using single-threaded versions of Orbix 1.3 and ORBeline 1.2. At the time these tests were performed, neither Orbix nor ORBeline fully supported the OMG 2.0 CORBA standard. This complicated the CORBA versions of `ttcp` somewhat since different implementations were required to account for differences in Orbix and ORBeline.

Extending ttcp to use CORBA required several modifications to the original C/socket code. All C socket calls were replaced with stubs and skeletons generated from pair of CORBA interface definitions. One IDL interface uses a sequence to transmit the data and the other IDL interface uses a string, as follows:

```
typedef sequence<char> ttcp_sequence;

interface TTCP_Sequence
{
  oneway void send (in ttcp_sequence ttcp_seq);
};

interface TTCP_String
{
  oneway void send (in string ttcp_string);
};
```

The send operations use oneway semantics since the ttcp benchmarks measure the performance of uni-directional data transfers.

The client-side of ttcp was modified to obtain object references to the server-side TTCP_Sequence and TTCP_String object implementations, as follows:

```
// Use locator service to acquire bindings.
TTCP_String *t_str = TTCP_String::_bind ();
TTCP_Sequence *t_seq = TTCP_Sequence::_bind ();
```

Data buffers of the appropriate size were initialized and then transmitted by calling the IDL-generated send stubs, as follows:

```
// String transfer.

char *buffer = new char[buffer_size];
// Initialize data in char * buffer...

while (--buffers_sent >= 0)
  t_str->send (buffer);

// Sequence transfer.

TTCP_Sequence sequence_buffer;
// Initialize data in TTCP_Sequence buffer...

while (--buffers_sent >= 0)
  t_seq->send (sequence_buffer);
```

The server-side was modified to create object implementations for TTCP_Sequence and TTCP_String. CORBA IDL compilers generate skeletons that translate IDL interface definitions (such as TTCP_Sequence) into C++ base classes (such as TTCP_SequenceBOAImpl). Each IDL operation (such as oneway void send) is mapped to a corresponding C++ pure virtual method (such as virtual void send). Programmers then define C++ derived classes that override these virtual methods to implement application-specific functionality, as follows:[1]

[1]Both CORBA implementations of ttcp used inheritance since ORBeline does not support Orbix's "TIE" technique (which uses object composition to tie application-specific classes to the generated IDL skeletons).

```
// Implementation class for IDL interface
// that inherits from automatically-generated
// CORBA skeleton class.

class TTCP_Sequence_i
  : virtual public TTCP_SequenceBOAImpl
{
public:
  TTCP_Sequence_i (void): nbytes_ (0) {}

  // Upcall invoked by the CORBA skeleton.
  virtual void send
    (const TTCP_Sequence &ttcp_seq,
     CORBA::Environment &IT_env)
  {
    this->nbytes_ += ttcp_seq._length;
  }
  // ...

private:
  // Keep track of bytes received.
  u_long nbytes_;
};
```

The server-side used the CORBA impl_is_ready event loop to demultiplex incoming requests to the appropriate object implementation, as follows:

```
int main (int argc, char *argv[])
{
  // Implements the Sequence object.
  TTCP_Sequence_i ttcp_sequence;

  // Implements the String object.
  TTCP_String_i ttcp_string;

  // Tell the ORB that the objects are active.
  CORBA::BOA::impl_is_ready ();

  /* NOTREACHED */
  return 0;
}
```

Porting ttcp to use CORBA over ATM demonstrated the importance of having sufficient hooks to manipulate underlying OS mechanisms (such as transport layer and socket layer options) that significantly affect performance. In particular, high performance data transfers over TCP and ATM require large socket queues. This is illustrated by the considerable difference in throughput for the 8K and 64K socket queues in Figures 2 and 3.

Orbix provides hooks to enlarge socket queues via setsockopt by invoking a user-defined callback function whenever a new socket is connected. In contrast, it was hard to enlarge the socket queues using ORBeline 1.2 since it did not provide direct access to sockets (subsequent versions of ORBeline will provide this functionality).

By comparing Figure 3 with Figure 2 it is clear that the CORBA-based ttcp implementations ran considerably slower than the C and ACE wrapper versions on the ATM network, particularly for 8K data buffers. The highest throughput (39 Mbps) was obtained by the Orbix sequence implementation using 64K data buffers and 64K socket queues. The performance leveled off beyond 64K data buffers.

Unlike the C and ACE wrapper results in Figure 1, the performance of the CORBA versions did not decrease when the size of the data buffers exceeds 8K. This behavior stems from the higher fixed overhead of CORBA (such as demultiplexing and memory management) which lowers its performance for small buffer sizes. As the buffer size increases, however, the relative impact of this fixed overhead is reduced. However, as the buffers increase in size the overhead of data copying grows, which ultimately limits the throughput achievable with the CORBA implementations.

Further profiling and examination of the IDL stubs and skeletons generated by Orbix and ORBeline revealed that the CORBA overhead stems from the following sources:

• **Data Copying:** The data buffers exchanged between the sender and receiver in `ttcp` are treated as a stream of untyped bytes. This is similar to the type of data transmitted by streaming applications such as teleconferencing and medical imaging [13]. Since the data is untyped the CORBA presentation layer need not perform complex marshalling to handle byte-ordering differences between sender and receiver.

Although marshalling is not required, the CORBA implementations incurred significant data copying overhead. The UNIX profiler `prof` was used to pinpoint the sources of this overhead. `prof` measures the amount of time spent in functions during program execution. Figure 4 lists the functions for all the tests where the most time was spent sending and receiving 64 Mbytes using 128K data buffers and 64K socket queues:

The `read` and `write` system calls accounted for most of the execution time in the C and ACE wrapper implementations of `ttcp`. The remaining time for the sender-side was spent preparing the data for transmission. Note that although the data was transmitted as 512 128K buffers it was read by the receiver in much smaller chunks (around 4K). This illustrates the fragmentation and reassembly performed by the ATM network adaptors.

The `read` and `write` system calls dominated the execution of the CORBA implementations, as well. However, unlike the C and ACE wrapper versions, these implementations spent 4 to 15 percent of their time performing other tasks, such as copying and/or inspecting data (`memcpy`, `strcpy`, and `strlen`), checking for activity on other handles (`_poll`), and manipulating signal handlers (`__sigaction`).

The highest cost tasks involved data copying. The IDL stubs and sequences copy data multiple times, *e.g.*, from the TCP data buffer into a marshalling buffer, and then again into the parameter passed to the `send` upcall. The results in Figure 3 illustrate that the choice of CORBA IDL parameter datatypes has a significant impact on performance. The `sequence` implementations shown in Figure 3 peaked at 39 Mbps for Orbix and 38 Mbps for ORBeline. In contrast, the `string` implementations peaked at 34 Mbps for Orbix and 30 Mbps for ORBeline.

The performance variation between the `sequence` and `string` results from differences in their IDL to C++ mappings. In particular, the IDL `sequence` mapping contains a

Test	%Time	#Calls	msec/call	Name
C sockets (sender)	99.6	527	92.8	_write
C sockets (receiver)	99.3	14808	3.2	_read
ACE wrapper (sender)	99.4	527	87.3	_write
ACE wrapper (receiver)	99.6	14759	3.2	_read
Orbix Sequence (sender)	94.6	532	89.1	_write
	4.1	2121	1.0	memcpy
Orbix Sequence (receiver)	92.7	7860	6.1	_read
	4.8	2581	6.1	memcpy
Orbix String (sender)	89.0	532	85.6	_write
	4.6	2121	1.1	memcpy
	4.1	2700	0.7	strlen
Orbix String (receiver)	86.3	7744	5.7	_read
	5.5	6740	0.4	strlen
	4.5	2581	0.9	memcpy
ORBeline Sequence (sender)	91.0	551	74.9	_write
	5.2	6413	0.4	memcpy
	1.8	1032	0.8	__sigaction
ORBeline Sequence (receiver)	89.0	7568	5.8	_read
	5.1	7222	0.3	memcpy
	3.3	1071	1.5	_poll
ORBeline String (sender)	83.8	551	83.9	_write
	5.4	920	3.2	strcpy
	4.3	5901	0.4	memcpy
	3.9	1728	1.2	strlen
	1.1	1032	0.6	__sigaction
ORBeline String (receiver)	85.4	7827	5.5	_read
	4.6	6710	0.3	memcpy
	4.2	1702	1.3	strlen
	2.8	1071	1.3	_poll

Figure 4: High-cost Functions for `ttcp` Tests

length field, whereas the `string` mapping does not. Thus, the generated stubs and skeletons use this length field to avoid searching each `sequence` parameter for a terminating NUL character. In contrast, the IDL `string` implementations use `strlen` to determine the length of their parameters.

The performance variation between Orbix and ORBeline results from differences in their message fragmentation/reassembly implementation, as well as the design of their socket event handling. As shown in Figure 4, ORBeline copies data approximately 3 more times than Orbix on the sender and receiver for both `sequence` and `string`. In addition, ORBeline invokes the `poll` and `sigaction` system calls over 1,000 times. The Orbix implementation does not perform these extra operations, which is one reason why ORBeline performance consistently lower than Orbix in Figure 3.

• **Demultiplexing:** Each CORBA request message contains the name of its remote operation represented as a string. Orbix demultiplexes incoming messages to the upcall by performing a linear search through the list of operations in the IDL interface. In the case of `ttcp`, linear search suffices since there was only one choice (`send`). However, this strategy does not scale since search time grows linearly with the number of operations in the IDL interface. Moreover, the order of operations will determine the demultiplexing performance. Therefore, operations in Orbix should be ordered by decreasing frequency of use.

In contrast, ORBeline use hashing to determine the appropriate upcall associated with an incoming request. Hashing is likely to scale better for large IDL interfaces, but may be less efficient for small interfaces. Thus, demultiplexing may benefit from adaptive optimizations that select customized strategies depending on the properties of the IDL interface. Alternatively, perfect hashing [14] or some type of integral indexing scheme could be negotiated between sender and receiver to improve performance and to shield developers from having to manually tune their IDL interfaces.

• **Memory allocation:** CORBA-generated skeletons do not know how the user-supplied upcall will use the parameters passed to it from the request message. Thus, they use conservative memory management techniques that dynamically allocate and release copies of messages before and after an upcall, respectively. These memory management policies are important in some circumstances (*e.g.*, if an upcall is used in a multi-threaded application). However, this strategy needlessly increases processing overhead for streaming applications like `ttcp` that immediately consume their data without modifying it.

2.3 Evaluation and Recommendations

Section 2 compared the performance of C, ACE wrapper, and CORBA versions of `ttcp` in terms of their ability to stream large qualities of data using TCP/IP over Ethernet and ATM networks. Tables 1 and 2 summarize the results for all the ATM and Ethernet tests, respectively. All tests perform roughly the same on Ethernet. However, the data copying overhead of the CORBA implementations significantly limits their throughput on ATM. This illustrates that the overhead of CORBA implementations may not be revealed until the network is no longer the limiting factor. In addition, the profiler results in Figure 4 illustrate that small design and implementation differences have a large performance impact on high-speed networks.

As users and organizations migrate to high-speed networks the performance limitations of contemporary CORBA implementations will become more evident. This should encourage vendors to optimize the performance of their ORBs for streaming applications running over high-speed networks such as ATM. Key areas of optimization include presentation layer conversions, memory management and memory copying, and receiver-side demultiplexing and dispatching. In particular, implementations must reduce the number of times that large data buffers are copied on the sender and receiver. The need for these optimizations is widely recognized in the communication protocol community [12] and prototypes that implementation these optimizations are becoming available [15].

Until these optimizations are widely implemented in production systems, however, we recommend that developers of bandwidth-intensive and delay-sensitive streaming applications on high-speed networks consider the following when adopting a distributed object computing solution:

- Carefully measure the performance of the communication infrastructure (*i.e.*, the network/host hardware and software). The `ttcp` benchmarks and ACE source code described in this paper are freely available and may be obtained via anonymous ftp from `ics.uci.edu` in the file `gnu/C++_wrappers.tar.Z` or from URL `http://www.cs.wustl.edu/~schmidt/`. We encourage others to replicate our `ttcp` experiments using different implementations of CORBA and other network/host platforms and report the results.
- Evaluate tools based on empirical measurements and thorough understanding of application requirements, rather than adopting a particular communication model or implementation unconditionally.
- Insist that CORBA implementors provide hooks to manipulate the underlying protocol layer and socket layer options conveniently. It is particularly important to increase the size of the socket queues to the largest values supported by the OS.
- Tune the size of transmitted data buffers to match the MTU of the network where appropriate.
- Use IDL `sequences` rather than `strings` to avoid unnecessary data access.

The performance results and recommendations in this paper are not intended as a criticism of the CORBA model or of particular ORB vendors. It is beyond the scope of

Program	Buffer	1 Byte	1K	2K	4K	8K	16K	32K	64K	128K
C Code	64K	0.16	23.52	32.19	43.16	61.77	58.10	56.13	48.91	48.6
	8K	0.16	20.77	22.19	23.30	23.57	23.22	22.43	22.63	22.69
ACE Wrapper	64K	0.14	22.29	31.85	42.54	61.81	60.41	56.65	47.67	47.90
	8K	0.14	20.48	22.13	23.66	23.12	23.85	23.58	23.63	23.72
Orbix (Sequence)	64K	0.01	7.85	12.72	18.94	25.79	31.65	34.87	39.15	38.44
	8K	0.01	7.71	11.58	16.60	21.90	22.81	21.89	21.01	20.98
Orbix (String)	64K	0.01	8.15	13.18	19.02	23.02	29.71	30.98	33.78	32.23
	8k	0.01	8.43	11.62	16.61	22.11	22.76	21 .27	20.07	20.00
ORBeline (Sequence)	64K	0.01	6.06	12.14	24.36	26.13	35.23	36.70	37.67	36.72
	8K	0.01	10.61	16.09	19.44	21.83	22.75	22.45	21.45	21.05
ORBeline (String)	64K	0.01	9.35	14.97	22.02	22.91	28.99	29.58	28.66	28.42
	8K	0.01	10.22	17.14	20.61	21.16	21.93	19.84	18.28	18.10

Table 1: Results for ATM `ttcp` Tests

Program	Buffer	1 Byte	1K	2K	4K	8K	16K	32K	64K	128K
C Code	64K	0.12	8.30	8.46	8.67	8.78	8.66	8.67	8.68	8.71
	8K	0.12	8.10	8.21	8.33	8.30	8.22	8.32	8.36	8.37
ACE Wrapper	64K	0.12	8.22	8.34	8.74	8.72	8.61	8.65	8.68	8.70
	8K	0.12	7.97	8.06	8.38	8.31	8.28	8.19	8.31	8.41
Orbix (Sequence)	64K	0.01	6.68	8.30	8.52	8.51	8.45	8.47	8.44	8.38
	8K	0.01	6.66	7.80	7.97	8.18	8.11	8.20	8.29	8.25
Orbix (String)	64K	0.01	6.42	8.36	8.55	8.66	8.59	8.58	8.52	8.45
	8k	0.01	6.47	7.82	7.85	8.10	8.17	8.23	8.34	8.30
ORBeline (Sequence)	64K	0.01	8.14	8.37	8.63	8.58	8.61	8.64	8.79	8.38
	8K	0.01	7.28	7.70	7.99	8.02	8.21	8.30	8.20	8.22
ORBeline (String)	64K	0.01	8.02	8.44	8.68	8.65	8.67	8.70	8.72	8.29
	8K	0.01	7.40	7.56	7.85	8.00	8.05	8.04	7.99	8.01

Table 2: Results for Ethernet `ttcp` Tests

this paper to discuss the benefits (such as extensibility and maintainability) of CORBA, as well as its limitations [16]. Clearly, implementations of other DOC frameworks (such as OODCE or OLE/COM) that do not address key sources of overhead on high-speed networks will exhibit similar performance problems.

3 An Object-Oriented Network Programming Interface

Low-level network programming interfaces like sockets or TLI are difficult to program. They require strict attention to many tedious details, making them hard to learn and error prone to program. In addition, programming directly to low-level interfaces limits portability and reuse.

One solution is to develop applications using higher-level distributed object computing (DOC) frameworks like CORBA. DOC frameworks shield developers from low-level programming details and facilitate a reasonably portable distributed computing platform. As described in the previous section, however, the performance of conventional implementations of CORBA may be inadequate for bandwidth-intensive and delay-sensitive streaming applications on high-speed networks.

One method for satisfying the tension between programming simplicity, portability, and run-time efficiency is to encapsulate lower-level network programming interfaces with object-oriented wrappers. By judicious use of languages features (such as inlining and templates) and design patterns (such as Factories [17], Connectors and Acceptors [18]) it is possible to create reusable object-oriented components that are typesafe, portable, convenient to program, *and* efficient.

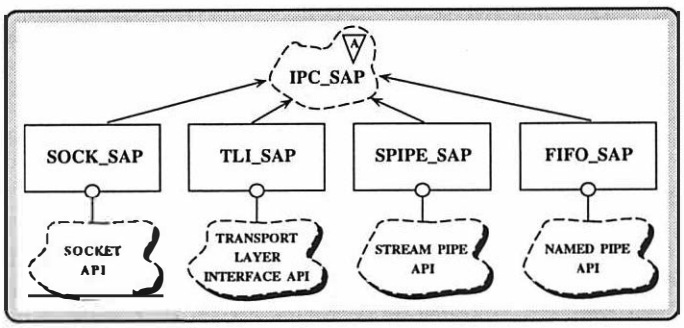


Figure 5: IPC SAP Class Category Relationships

This section outlines the design of the `IPC SAP` object-oriented network programming components provided by the ACE toolkit [6]. ACE contains a set of object-oriented networking programming components that perform active and passive connection establishment, data transfer, event demultiplexing, event handler dispatching, routing, dynamic (re)configuration of application services, and concurrency control [6].

`IPC SAP` stands for "InterProcess Communication Service Access Point." It consists of a family of class categories

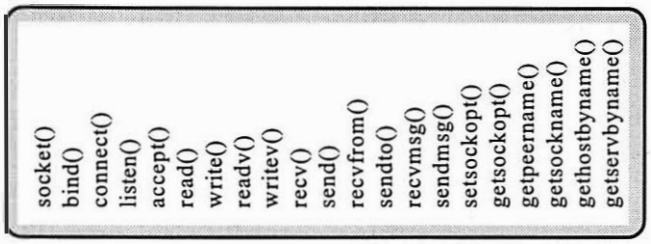


Figure 6: Socket Interface

shown in Figure 5 that encapsulate handle-based network programming interfaces such as sockets (SOCK SAP), TLI (TLI SAP), UNIX SVR4 STREAM pipes (SPIPE SAP), and UNIX named pipes (FIFO SAP). These network programming wrappers are designed to improve the correctness, programming simplicity, portability, and reusability of performance-sensitive communication software. This section describes the SOCK SAP socket wrappers, focusing on interface design techniques that shield programmers from shortcomings of C, C++, and existing OS network programming interfaces.

3.1 Limitations with Sockets

Sockets were originally developed in BSD UNIX to provide an interface to the TCP/IP protocol suite [4]. From an application's perspective, a socket is a local endpoint of communication that can be bound to an address residing on a local or a remote host. Sockets are accessed via *handles*, which are unsigned integers that index into a table maintained in the OS. Handles shield applications from the internal representation of OS data structures. In UNIX and Windows NT, socket handles share the same name space as other handles (such as files, named pipes, and terminal devices).

The standard socket interface is defined by the C functions shown in Figure 6. It contains several dozen routines that perform tasks such as locating address information for network services, establishing and terminating connections, and sending and receiving data [19]. Although the socket interface is widely available and widely used, its design has several notable limitations discussed below. These limitations are shared by other lower-level network programming interfaces such as TLI, STREAM pipes, and named pipes.

3.1.1 High Potential for Error

In UNIX any integral value can be passed as a handle to a socket routine. Therefore, compilers are unable to detect or prevent the erroneous use of handles. This weak type-checking allows subtle errors to occur at run-time since the socket interface cannot enforce the correct use of routines for different communication roles (such as active vs. passive connection establishment or datagram vs. stream communication). Operations (such as invoking a data transfer operation on a handle designated for establishing connections) may therefore be applied improperly on handles.

Figure 7 depicts the following subtle (and common) errors that occur when using the socket interface:

1. Forgetting to initialize the `length` parameter (used by `accept`) to the size of `struct sockaddr_in`;
2. Forgetting to "zero-out" all bytes in the socket address structure;
3. Using an address family type that is inconsistent with the protocol family of the socket (*e.g.*, PF_UNIX vs. AF_INET);
4. Neglecting to use the `htons` library function to convert port numbers from host byte-order to network byte-order and vice versa;
5. Applying the `accept` function on a SOCK_DGRAM socket;
6. Erroneously omitting parentheses in an assignment expression;
7. Trying to `read` from a passive-mode socket that should only be used to `accept` connections;
8. Failing to properly detect and handle "short-writes" that occur due to buffering in the OS and flow control in the transport protocol.

Other common misuses of sockets not shown in this example are forgetting to call `listen` when creating a passive-mode SOCK_STREAM listener socket and miscalculating the length of the pathname in a UNIX-domain socket address (the trailing NUL should not be counted).

Several of the problems listed above are classic problems with programming in C. For instance, by omitting the parentheses in the expression

```
if (n_fd = accept (s_fd,
                   (struct sockaddr *) &s_addr,
                   &length) == -1)
```

the value of `n_fd` will always be set to either 0 or 1, depending on whether `accept() == -1`. This problem is exacerbated by the fact that `accept` returns the handle of the newly connected socket. If this handle were passed back as an out parameter there would be less incentive to use `accept` in an assignment expression.

A deeper problem is that C's lack of support for data abstraction and object-oriented programming makes it hard to define typesafe, extensible, and reusable component interfaces. For example, the generic `sockaddr` socket address structure provides a crude form of inheritance to express the commonality between Internet domain and UNIX domain address structures (`sockaddr_in` and `sockaddr_un`, respectively). These "subclass" address structures require the use of a non-typesafe cast to overlay the `sockaddr` "base class." In an object-oriented language this commonality would be expressed more cleanly and robustly using inheritance and dynamic binding.

In general, the use of unsafe typecasts, combined with the weakly-typed handle-based socket interface, makes it impossible for a compiler to detect mistakes at compile-time.

```
int echo_server (u_short port_num)
{
  sockaddr_in s_addr;
  int length; // (1) uninitialized variable.
  char buf[BUFSIZ];
  int s_fd, n_fd;
  // Create a local endpoint of communication.
  s_fd = socket (PF_UNIX, SOCK_DGRAM, 0);

  // Set up address information to become a server.
  // (2) forgot to "zero out" structure first...

  // (3) used the wrong address family ...
  s_addr.sin_family = AF_INET;

  // (4) forgot to use htons() on port_num...
  s_addr.sin_port = port_num;
  s_addr.sin_addr.s_addr = INADDR_ANY;

  bind (s_fd, (struct sockaddr *) &s_addr,
        sizeof s_addr) == -1)

  // Create a new endpoint of communication.
  // (5) can't accept() on a SOCK_DGRAM.
  // (6) Omitted a crucial set of parens...
  if (n_fd = accept (s_fd,
                     (struct sockaddr *) &s_addr,
                     &length) == -1) {
    int n;
    // (6) Omitted another set of parens...
    // (7) can't read from s_fd.
    while (n = read (s_fd, buf, sizeof buf) > 0)

      // (8) forgot to check for "short-writes"
      write (n_fd, buf, n);
    // Remainder omitted...
  }
}
```

Figure 7: Socket version of Echo Server

Instead, error checking is deferred until run-time, which complicates error handling and reduces application robustness.[2]

3.1.2 Complex Interface

Sockets support multiple protocol families (such as TCP/IP, IPX/SPX, ISO OSI, and UNIX domain sockets) with a single interface. The socket interface contains many functions to support different *communication roles* (such as active vs. passive connection establishment), *communication optimizations* (such as `writev/readv` that send/receive multiple buffers in a single system call), and *protocol options* (such as broadcasting, multicasting, asynchronous I/O, and urgent data delivery).

Although sockets combine this functionality into a common interface, the result is complex and hard to master. Much of this complexity stems from the overly broad and one-dimensional design of the socket interface. That is, all the routines appear at a single level of abstraction (as shown in Figure 6). This design increases the amount of effort required to learn and use sockets correctly. In particular, programmers must understand most of the interface to use any part of it effectively.

If the socket routines are examined carefully, however, it is clear that the interface decomposes naturally into the following communication dimensions:

1. *Type of communication service – i.e.*, stream vs. datagram vs. connected datagram;
2. *Communication role – i.e.*, active vs. passive (clients are typically active, whereas servers are typically passive);
3. *Communication domain – i.e.*, local IPC only vs. local/remote IPC.

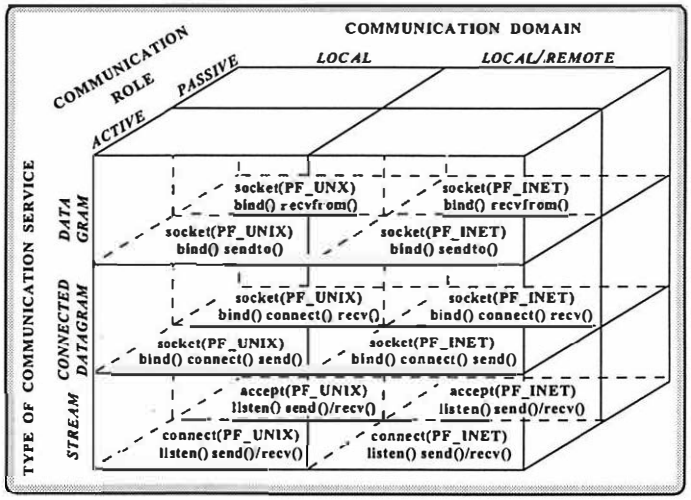


Figure 8: A Taxonomy of Socket Communication Dimensions

Figure 8 classifies the socket routines according to these dimensions. Since the socket interface is one-dimensional, however, this natural clustering of functionality is obscured.

Another problem with the socket interface is that its several dozen routines lack a uniform naming convention. Non-uniform naming makes it hard to determine the scope of the socket interface. For example, it is not immediately obvious that `socket`, `bind`, `accept`, and `connect` routines are related. Other network programming interfaces solve this problem by prepending a common prefix before each routine. For example, a `t_` is prepended before each routine in the TLI library.

3.2 The SOCK SAP Class Category

`SOCK SAP` is designed to overcome the limitations with sockets described above. It improves the correctness, ease of learning and ease of use, reusability, and portability of communication software without sacrificing performance. This section outlines the software architecture of `SOCK SAP` and explains the classes used by the programming examples in Section 4. Readers who are not interested in this level of detail may want to skip to Section 5, which discusses the general principles underlying the design of the `SOCK SAP` wrappers.

[2] Most of the error checking has been omitted in these examples to save space. Naturally, robust programs should check the return values of library and system calls.

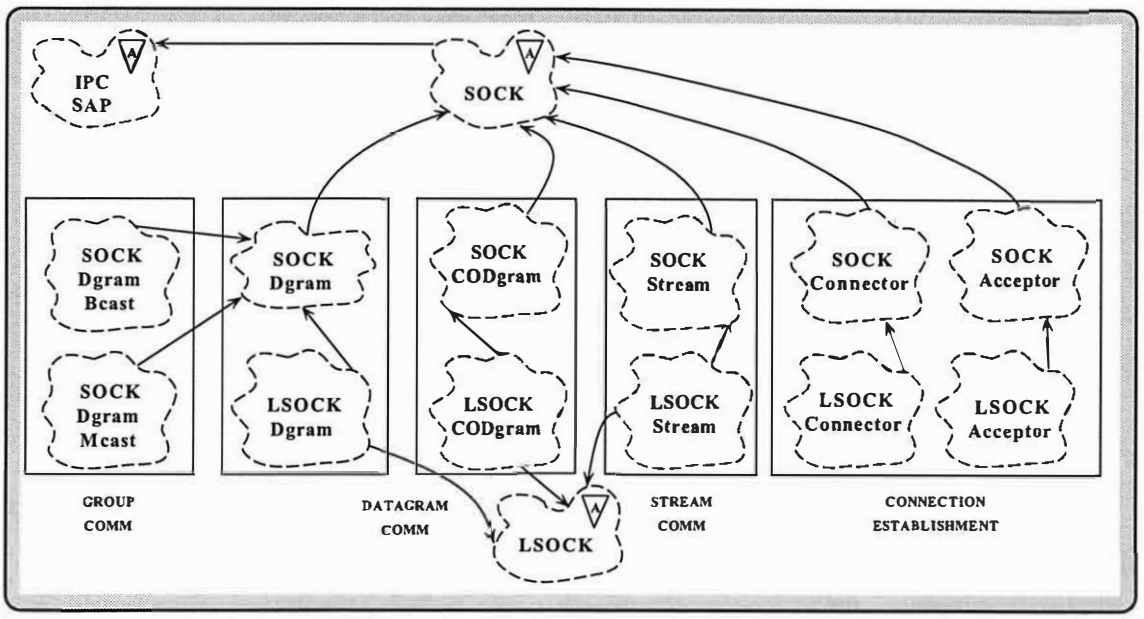


Figure 9: The SOCK SAP Class Categories

SOCK SAP consists of around one dozen C++ classes that are related by multiple inheritance and composition. These components and their relationships are illustrated via Booch notation [20] in Figure 9. Dashed clouds indicate classes and directed edges indicate inheritance relationships between these classes (*e.g.*, SOCK Stream inherits from SOCK). The general structure of SOCK SAP corresponds to the taxonomy of *communication services*, *communication roles*, and *communication domains* shown in Figure 10. It is instructive to compare Figure 8 with Figure 10. The latter is more concise since it uses C++ wrappers to encapsulate the behavior of multiple socket mechanisms within classes related by inheritance.

Each class in SOCK SAP provides an abstract interface for a subset of mechanisms that together comprise the overall class category. The functionality of various types of Internet-domain and UNIX-domain sockets is achieved by inheriting mechanisms from the appropriate classes described below. These classes are presented below according to the groupings shown in Figure 9.

Base Classes: The SOCK and LSOCK classes anchor the inheritance hierarchy and enable subsequent derivation and code sharing. Objects of these classes cannot be instantiated since their constructors are declared in the protected section of the class definition.

- **SOCK:** this class is the root of the SOCK SAP hierarchy. It provides mechanisms common to all other classes, such as opening and closing local endpoints of communication and handling options (such as selecting socket queue sizes and enabling group communication).

- **LSOCK:** this class provides mechanisms that allow applications to send and receive open file handles between unrelated processes on the local host machine (hence the prefix 'L'). Note that System V and BSD UNIX both support this feature, though Windows NT does not. Other classes inherit from LSOCK to obtain this functionality.

SOCK SAP distinguishes the LSOCK* and SOCK* classes on the basis of network address formats and communication semantics. In particular, the LSOCK* classes use UNIX pathnames as addresses and only allow intra-machine IPC. The SOCK* classes, on the other hand, use Internet Protocol (IP) addresses and port numbers and allow both intra- and inter-machine IPC.

Connection Establishment: Communication software is typified by asymmetric connection behavior between clients and servers. In general, servers listen *passively* for clients to initiate connections *actively* [18]. The structure of passive/active connection establishment and data transfer relationships are captured by the following connection-oriented SOCK SAP classes:

- **SOCK Acceptor and LSOCK Acceptor:** The *Acceptor classes are factories [17] that passively establish new endpoints of communication in response to active connection requests. The SOCK Acceptor and LSOCK Acceptor factories produce SOCK Stream and LSOCK Stream connection endpoint objects, respectively.

- **SOCK Connector and LSOCK Connector:** The *Connector classes are factories that actively establish new endpoints of communication. These classes establish connections with remote endpoints and produce the appropriate *Stream object when a connection is established. A connection may be initiated either synchronously or asynchronously. The SOCK Connector and LSOCK

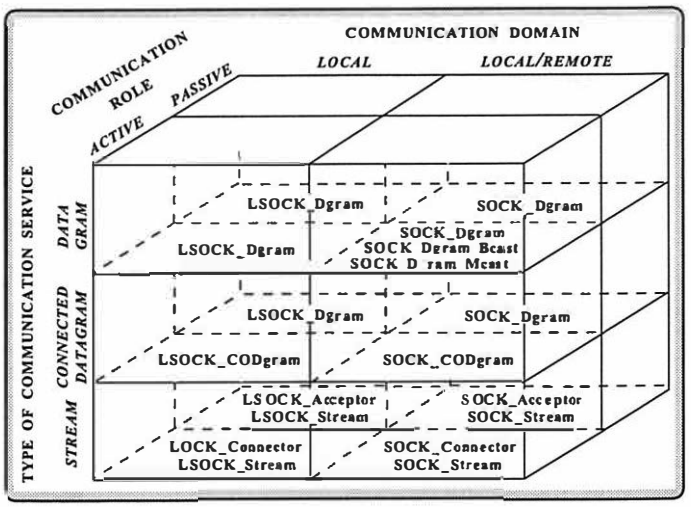


Figure 10: Taxonomy of SOCK SAP Classes and Communication Dimensions

`Connector` factories produce `SOCK Stream` and `LSOCK Stream` connection endpoint objects, respectively.

Note that the `*Acceptor` and `Connector` classes do not provide methods for sending or receiving data. Instead, they are factories that produce the `*Stream` data transfer objects described below. The use of strongly-typed interfaces detects accidental misuse of local and non-local `*Stream` objects at compile-time. In contrast, the socket interface can only detect these type mismatches at run-time.

Stream Communication: Although establishing connections requires a distinction between active and passive roles, once a connection is established data may be exchanged in any order according to the protocol used by the endpoints. `SOCK SAP` isolates the data transfer behavior in the following classes:

- **SOCK Stream and LSOCK Stream:** These two classes are produced by the `*Acceptor` or `*Connector` factories described above. The `*Stream` classes provide mechanisms for transferring data between two processes. `LSOCK Stream` objects exchange data between processes on the same host machine; `SOCK Stream` objects exchange data between processes that may reside on different host machines.

The overloaded `send` and `recv` `*Stream` methods provide standard UNIX `write` and `read` semantics. Thus, a `send` may write less (and a `recv` may read more) than the requested number of bytes. These "short-writes" and "short-reads" occur due to buffering in the OS and flow control in the transport protocol. To reduce programming effort, the the `*Stream` classes provide `send_n` and `recv_n` methods that allow transmission and reception of exactly n bytes. "Scatter-read" and "gather-write" methods are also provided to efficiently send and receive multiple buffers of data simultaneously.

Datagram Communication:

- **SOCK CODgram and LSOCK CODgram:** These classes provide a "connected-datagram" mechanism, which allows the `send` and `recv` operations to omit the address of the service when exchanging datagrams. Note that the connected-datagram mechanism is only a syntactic convenience since there are no additional semantics associated with the data transfer (*i.e.*, datagram delivery remains unreliable). `SOCK CODgram` inherits mechanisms from the `SOCK` base class. `LSOCK CODgram` inherits mechanisms from both `SOCK CODgram` and `LSOCK` (which provides the ability to pass file handles).

- **SOCK Dgram and LSOCK Dgram:** These classes provide mechanisms for exchanging datagrams between processes running on local and/or remote hosts. Unlike the connected-datagram classes described above, each `send` and `recv` operation *must* provide the address of the service with every datagram sent or received. `LSOCK Dgram` inherits all the operations of both `SOCK Dgram` and `LSOCK`. It only exchanges datagrams between processes on the same host. The `SOCK Dgram` class, on the other hand, may exchange datagrams between processes on local and/or remote hosts.

Group Communication:

- **SOCK Dgram Bcast:** This class provides mechanisms for broadcasting UDP datagrams to processes running on local and/or remote hosts attached to local subnets. The interface for this class supports the broadcast of datagrams to (1) all network interfaces connected to the host machine or (2) a particular network interface. This class shields the end-user from the low-level details required to utilize broadcasting effectively.

- **SOCK Dgram Mcast:** This class provides mechanisms for multicasting UDP datagrams to processes running on local and/or remote hosts attached to local subnets. The interface for this class supports the multicast of datagrams to a particular multicast group. This class shields the end-user from the low-level details required to utilize multicasting effectively.

4 Programming with SOCK SAP C++ Wrappers

This section illustrates the ACE `SOCK SAP` wrappers by using them to develop a client/server streaming application. This application is simplified version of the `ttcp` program described in Section 2. For comparison, this application is also written with sockets and CORBA.

```
#define PORT_NUM 10000
#define TIMEOUT 5

/* Socket client. */

void send_data (const char host[],
                u_short port_num)
{
  struct sockaddr_in peer_addr;
  struct hostent *hp;
  char buf[BUFSIZ];
  int s_fd, w_bytes, r_bytes, n;

  /* Create a local endpoint of communication */
  s_fd = socket (PF_INET, SOCK_STREAM, 0);

  /* Set s_fd to non-blocking mode. */
  n = fcntl (s_fd, F_GETFL, 0);
  fcntl (s_fd, F_SETFL, n | O_NONBLOCK);

  /* Determine IP address of the server */
  hp = gethostbyname (host);

  /* Set up address information to contact server */
  memset ((void *) &peer_addr, 0, sizeof peer_addr);
  peer_addr.sin_family = AF_INET;
  peer_addr.sin_port = port_num;
  memcpy (&peer_addr.sin_addr,
          hp->h_addr, hp->h_length);

  /* Establish non-blocking connection server. */
  if (connect (s_fd, (struct sockaddr *) &peer_addr,
               sizeof peer_addr) == -1) {
    if (errno == EINPROGRESS) {
      struct timeval tv = {TIMEOUT, 0};
      fd_set rd_fds, wr_fds;
      FD_ZERO (&rd_fds);
      FD_ZERO (&wr_fds);
      FD_SET (s_fd, &wr_fds);
      FD_SET (s_fd, &rd_fds);

      // Wait for up to TIMEOUT seconds to connect.
      if (select (s_fd + 1, &rd_fds, &wr_fds,
                  0, &tv) <= 0)
        perror ("connection timedout"), exit (1);
      if (connect (s_fd,
                   (struct sockaddr *) &peer_addr,
                   sizeof peer_addr) == -1
          && errno != EISCONN)
        perror ("connect failed"), exit (1);
  }

  /* Send data to server (correctly handles
     "short writes" due to flow control) */

  while ((r_bytes = read (0, buf, sizeof buf)) > 0)
    for (w_bytes = 0; w_bytes < r_bytes; w_bytes += n)
      n = write (s_fd, buf + w_bytes,
                 r_bytes - w_bytes);

  /* Close down the connection. */
  close (s_fd);
}

int main (int argc, char *argv[])
{
  char *host = argc > 1 ? argv[1] : "ics.uci.edu";
  u_short port_num =
    htons (argc > 2 ? atoi (argv[2]) : PORT_NUM);

  // Send data to the server.
  send_data (host, port_num);
  return 0;
}
```

Figure 11: Socket-based Client Example

```
#define PORT_NUM 10000

/* Socket server. */

void recv_data (u_short port_num)
{
  struct sockaddr_in s_addr;
  int s_fd;

  /* Create a local endpoint of communication */
  s_fd = socket (PF_INET, SOCK_STREAM, 0);

  /* Set up the address information for a server */
  memset ((void *) &s_addr, 0, sizeof s_addr);
  s_addr.sin_family = AF_INET;
  s_addr.sin_port = port_num;
  s_addr.sin_addr.s_addr = INADDR_ANY;

  /* Associate address with endpoint */
  bind (s_fd, (struct sockaddr *) &s_addr,
        sizeof s_addr);

  /* Make endpoint listen for service requests */
  listen (s_fd, 5);

  /* Performs the iterative server activities */

  for (;;) {
    char buf[BUFSIZ];
    int r_bytes, n_fd;
    struct sockaddr_in peer_addr;
    int peer_addr_len = sizeof peer_addr;
    struct hostent *hp;

    /* Create a new endpoint of communication */
    while ((n_fd = accept (s_fd, &peer_addr,
                           &peer_addr_len)) == -1
           && errno == EINTR)
      continue;

    hp = gethostbyaddr (&peer_addr.sin_addr,
                        peer_addr_len, AF_INET);

    printf ("client %s\n", hp->h_name);

    /* Read data from client (terminate on error) */

    while ((r_bytes = read (n_fd, buf, sizeof buf)) > 0)
      write (1, buf, r_bytes);

    /* Close the new endpoint
       (listening endpoint remains open) */
    close (n_fd);
  }
  /* NOTREACHED */
}

int main (int argc, char *argv[])
{
  u_short port_num =
    htons (argc > 1 ? atoi (argv[1]) : PORT_NUM);

  // Receive data from clients.
  recv_data (port_num);
  return 0;
}
```

Figure 12: Socket-based Server Example

```
static const PORT_NUM = 10000;
static const TIMEOUT = 5;

// SOCK_SAP Client.

template <class PEER_CONNECTOR,
          class PEER_STREAM,
          class ADDR>
void send_data (ADDR peer_addr)
{
  // Data transfer object.
  PEER_STREAM peer_stream;

  // Establish connection without blocking.
  PEER_CONNECTOR connector
    (peer_stream, peer_addr, ACE_NONBLOCK);

  if (peer_stream.get_handle () == -1) {
    // If non-blocking connection is in progress,
    // wait up to TIMEOUT seconds to complete.
    Time_Value timeout (TIMEOUT);

    if (errno != EWOULDBLOCK ||
        connector.complete
          (peer_stream, peer_addr, &timeout) == -1)
      perror ("connector"), exit (1);
  }

  // Send data to server (send_n() handles
  // "short writes" correctly).

  char buf[BUFSIZ];

  for (int r_bytes;
       (r_bytes = read (0, buf, sizeof buf)) > 0;)
    peer_stream.send_n (buf, r_bytes);

  // Explicitly close the connection.
  peer_stream.close ();
}

int main (int argc, char *argv[])
{
  char *host = argc > 1 ? argv[1] : "ics.uci.edu";
  u_short port_num =
    htons (argc > 2 ? atoi (argv[2]) : PORT_NUM);

  // Address of the server.
  INET_Addr s_addr (port_num, host)

  // Use TLI wrappers on client's side.
  send_data <TLI_Connector, TLI_Stream,
             INET_Addr> (s_addr);
  return 0;
}
```

Figure 13: SOCK SAP-based Client Example

```
static const int PORT_NUM = 10000;

// SOCK_SAP Server.

template <class PEER_ACCEPTOR,
          class PEER_STREAM,
          class ADDR>
void recv_data (ADDR s_addr)
{
  // Factory for passive connection establishment.
  PEER_ACCEPTOR acceptor (s_addr);

  // Data transfer object.
  PEER_STREAM peer_stream;

  // Remote peer address.
  ADDR peer_addr;

  // Performs iterative server activities.

  for (;;) {
    // Create a new PEER_STREAM endpoint
    // (automatically restarted if errno == EINTR).
    acceptor.accept (peer_stream, &peer_addr);

    printf ("client %s\n", peer_addr.get_host_name ());

    // Read data from client (terminate on error).

    char buf[BUFSIZ];

    for (int r_bytes;
         peer_stream.recv (buf, sizeof buf,
                           r_bytes) > 0;)
      write (1, buf, r_bytes);

    // Close peer_stream endpoint
    // (acceptor endpoint stays open).
    peer_stream.close ();
  }
  /* NOTREACHED */
}

int main (int argc, char *argv[])
{
  u_short port_num =
    argc == 1 ? PORT_NUM : atoi (argv[1]);

  // Port for the server.
  INET_Addr s_addr (port_num);

  // Use socket wrappers on server's side.
  recv_data<SOCK_Acceptor, SOCK_Stream,
            INET_Addr> (s_addr);
  return 0;
}
```

Figure 14: SOCK SAP-based Server Example

```
// CORBA IDL interface.

interface Data_Stream
{
  typedef sequence<char> Stream_Buf;

  exception Disconnected {};

  oneway void send (in Stream_Buf buf)
    raises (Disconnected);
};

// CORBA Client.

void send_data (Data_Stream *peer_stream)
{
  // Constructor allocates memory.
  Data_Stream::Stream_Buf buf (BUFSIZ);

  // Read from stdin and send to server.

  while ((buf._length =
          read (0, buf._buffer, BUFSIZ)) > 0)
    peer_stream->send (buf);

  peer_stream->_release ();
}

int main (int argc, char *argv[])
{
  char *marker
    = argc > 1 ? argv[1] : "data_stream";
  Data_Stream *peer_stream =
    Data_Stream::_bind (marker);

  send_data (peer_stream);
  return 0;
}
```

Figure 15: CORBA-based Client Example

Figures 11 and 12 present a client/server program that uses Internet-domain sockets to implement the stream application. The server shown in Figure 12 creates a passive-mode listener socket and waits for clients to connect to it. Once connected, the server receives the data transmitted from the client and displays the data on its standard output stream. The client-side shown in Figure 11 establishes a TCP connection with the server and transmits its standard input stream across the connection. The client uses non-blocking connections to limit the amount of time it waits for a connection to be accepted or refused.

Most of the error checking for return values has been omitted to save space. However, it is instructive to note all the socket initialization, network addressing, and flow control details that must be programmed explicitly to make even this simple example work correctly. Moreover, the code in Figures 11 and 12 is not portable to platforms that do not support sockets or `select`.

Figures 13 and 14 use SOCK_SAP to reimplement the C versions of the client/server programs. The SOCK_SAP programs implement the same functionality as those presented in Figure 11 and Figure 12. The SOCK_SAP C++ programs exhibit the following benefits compared with the socket-based C implementation:

- *Decreased program size – e.g.*, a substantial reduction in the lines of code results from localizing active and pas-

```
// Implementation class for IDL interface
// that inherits from automatically-generated
// CORBA skeleton class.

class Data_Stream_i
  : public Data_StreamBOAImpl
{
  // Upcall invoked by the CORBA skeleton.
  virtual void send
    (const Data_Stream::Stream_Buf &,
     CORBA::Environment &);
};

// Upcall invoked by the CORBA skeleton.

void
Data_Stream_i::send
  (const Data_Stream::Stream_Buf &buf,
   CORBA::Environment &IT_env)
{
  // Write data to standard output.
  write (1, buf._buffer, buf._length);
}

// CORBA persistent server.

int main (int argc, char *argv[])
{
  char *executable = argv[0];
  char *marker
    = argc > 1 ? argv[1] : "data_stream";

  // Define an implementation object.
  Data_Stream_i data_stream (marker);

  // ACE method that registers service
  // with the ORB automatically.
  CORBA_Handler::activate_service
    ("Data_Stream", object_name, executable);

  // Tell the ORB that the objects are active.
  CORBA::Orbix.impl_is_ready ("Data_Stream");

  /* NOTREACHED */
  return 0;
}
```

Figure 16: CORBA-based Server Example

sive connection establishment in the SOCK Acceptor and SOCK Connector connection factories. In addition, default values are provided for constructor and method parameters, which reduces the number of arguments needed for common usage patterns.

- *Increased clarity – e.g.,* network addressing and host location is handled by the Addr class, which hides the subtle and error-prone details that must be programmed explicitly in Figures 11 and 12. Moreover, the low-level details of non-blocking connection establishment are performed by the SOCK Connector.
- *Increased typesafety – e.g.,* the SOCK Acceptor and SOCK Connector connection factory objects must be passed arguments of the SOCK Stream type. This prevents the type errors shown in Figure 7 from occurring at run-time.
- *Increased portability – e.g.,* switching between sockets and TLI simply requires changing

```
send_data <TLI_Connector, TLI_Stream,
           INET_Addr> (s_addr);
```

in the client to

```
send_data <SOCK_Connector, SOCK_Stream,
           INET_Addr> (s_addr);
```

and

```
recv_data<SOCK_Acceptor, SOCK_Stream,
          INET_Addr> (s_addr);
```

in the server to

```
recv_data<TLI_Acceptor, TLI_Stream,
          INET_Addr> (s_addr);
```

Conditional compilation directives can be used to further decouple the communication software from reliance upon a particular type of network programming interface.

However, the ACE wrappers share some of the same drawbacks as sockets. In particular, too much of the code required to program at this level is not directly related to the application. In contrast, Figures 15 and 16 illustrate the CORBA version of the stream application implemented using Orbix 1.3. This implementation is considerably more concise than both the C and ACE wrapper versions. CORBA performs the low-level communication details associated with service location, passive and active connection establishment, message framing, marshalling and demarshalling, demultiplexing, and upcall dispatching. This allows developers to concentrate on defining application-specific behavior, rather than wrestling with the details of network programming.

The persistent server shown in Figure 16 creates an implementation of a Data_Stream IDL interface and informs the ORB that it is ready to receive send requests. It uses a standard ACE class CORBA_Handler to register the server and object name with the Orbix daemon automatically.

The client shown in Figure 15 uses the Orbix locator service to bind to the marker exported by the Data_Stream server. Once bound, the client transmits all data from its standard input to the server via the Data_Stream::send proxy. This example is behaves slightly differently than the C and ACE wrapper versions since CORBA does not provide a standard means to obtain the host and port of the sender. Moreover, CORBA communication semantics are request-oriented rather than connection-oriented. Thus, other clients could conceivably bind to the same marker name and transmit data via its send method.

5 Socket Wrapper Design Principles

This section describes the design principles applied throughout the SOCK SAP class category. Although these principles are widely used in domains such as graphical user interfaces they are less widely applied in the communication software domain.

• **Only permit typesafe operations:** Several limitations with sockets discussed in Section 3.1 stem from the lack of typesafety in its interface. To enforce typesafety, SOCK SAP ensures all of its objects are properly initialized via constructors. In addition, to prevent accidental violations of typesafety, only legal operations are permitted on SOCK SAP objects. This latter point is illustrated in the SOCK SAP revision of echo_server shown in Figure 17. This version fixes the problems with sockets and C identified in Figure 7. Since SOCK SAP classes are strongly typed, invalid operations are rejected at compile-time rather than at run-time. For example, it is not possible to invoke recv or send on a SOCK_Acceptor connection factory since these methods are not part of its interface. Likewise, return values are used to convey success or failure of operations, rather than returning more detailed information. This reduces the potential for misuse in assignment expressions.

• **Simplify for the common case:** The key to this principle is "make it easy to use SOCK SAP correctly, hard to use it incorrectly, but not impossible to use it in ways the class designers did not anticipate originally." This principle is exemplified by the get_handle and set_handle methods provided by the IPC SAP root class. These methods extract and assign the underlying handle, respectively. By providing get_handle and set_handle, IPC SAP allows applications to circumvent its type-checking mechanisms in unforeseen situations where applications must interface directly with UNIX system calls (such as select) that expect a handle.

• **Define parsimonious interfaces:** This principle localizes the cost of using a particular abstraction. The IPC SAP interfaces limits the amount of details that application developers must remember. IPC SAP provides developers with distinct cluster of classes that perform various types of communication (such as connection-oriented vs. connectionless) and various roles (such as active vs. passive). For example, to reduce the change of error, the SOCK Acceptor class only permits operations that apply for programs playing passive roles and the SOCK Connector class only permits

```
int echo_server (u_short port_num)
{
  // Address of local server.
  INET_Addr s_addr (port_num);

  // Initialize the passive mode server.
  SOCK_Acceptor acceptor (s_addr);

  // Data transfer object.
  SOCK_Stream peer_stream;

  // Client remote address object.
  INET_Addr peer_addr;

  // Accept a new connection.
  if (acceptor.accept (peer_stream,
                       &peer_addr) != -1) {
    char buf[BUFSIZ];
    for (size_t n;
         peer_stream.recv (buf, sizeof buf, n) > 0;)
      // Handles "short-writes."
      if (peer_stream.send_n (buf, n) != n)
        // Remainder omitted.
  }
}
```

Figure 17: SOCK_SAP Revision of the Echo Server

operations that apply for programs playing an active role. In addition, sending and receiving open file handles has a much simpler calling interface using SOCK_SAP compared with using the highly-general UNIX sendmsg/recvmsg routines.

• **Replace one-dimensional interfaces with hierarchically-related class categories:** This principle involves using hierarchically-related class categories to restructure existing one-dimensional socket interfaces. The criteria used to structure the SOCK_SAP class category involved identifying and clustering related socket routines to maximize the reuse and sharing of class components.

Inheritance support different subsets of functionality for the SOCK_SAP class categories. For instance, not all operating systems support passing open file handles (*e.g.*, Windows NT). Thus, it is possible to omit the LSOCK class (described in Section 3.2) from the inheritance hierarchy without affecting the interfaces of other classes in the SOCK_SAP design.

Inheritance also increases code reuse and improves modularity. Base classes express *similarities* between class category components and derived classes express the *differences*. For example, the SOCK_SAP design places shared mechanisms towards the "root" of the inheritance hierarchy. These mechanisms include operations for opening/closing and setting/retrieving the underlying socket handles, as well as certain option management functions that are common to all the derived SOCK_SAP classes. Subclasses located towards the "bottom" of the inheritance hierarchy implement specialized operations that are customized for the type of communication provided (such as stream vs. datagram communication or local vs. remote communication). This approach avoids unnecessary duplication of code since the more specialized derived classes reuse the more general mechanisms provided at the root of the inheritance hierarchy.

```
template <class PEER_ACCEPTOR,
          class PEER_STREAM,
          class ADDR>
int echo_server (u_short port_num)
{
  // Local address of server.
  ADDR s_addr (port_num);

  // Initialize the passive mode server.
  PEER_ACCEPTOR acceptor (s_addr);

  // Data transfer object.
  PEER_STREAM peer_stream;

  // Remote address object.
  ADDR peer_addr;

  // Accept a new connection.
  if (acceptor.accept (peer_stream,
                       &peer_addr) != -1) {
    char buf[BUFSIZ];
    for (size_t n;
         peer_stream.recv (buf, sizeof buf,
                           n) > 0)
      if (peer_stream.send_n (buf, n) != n)
        // Remainder omitted.
  }
}

// Conditionally select IPC mechanism.
#if defined (USE_SOCKETS)
typedef SOCK_Stream PEER_STREAM;
typedef SOCK_Acceptor PEER_ACCEPTOR;
#else
typedef TLI_Stream PEER_STREAM;
typedef TLI_Acceptor PEER_ACCEPTOR;
#endif // MT_SAFE_SOCKETS.

const int PORT_NUM = 10000;

int main (void)
{
  // ...

  // Invoke the echo_server with appropriate
  // network programming interfaces.
  echo_server<PEER_ACCEPTOR,
              PEER_STREAM,
              INET_Addr> (PORT_NUM);
}
```

Figure 18: Template Version of the Echo Server

• **Enhance portability with parameterized types:** Wrapping sockets with C++ classes (rather than stand-alone C functions) helps to improve portability by allowing the wholesale replacement of network programming mechanisms via parameterized types. Parameterized types decouple applications from reliance on specific network programming interfaces. Figure 18 illustrates this technique by modifying the echo_server to become a C++ function template. Depending on certain properties of the underlying OS platform (such as whether it implements TLI or sockets more efficiently), the echo_server may be instantiated with either SOCK_SAP or TLI_SAP classes, as shown in Figure 18.

In general, the use of parameterized types is less intrusive and more extensible that conventional alternatives (such as implementing multiple versions or littering conditional compilation directives throughout the source code).

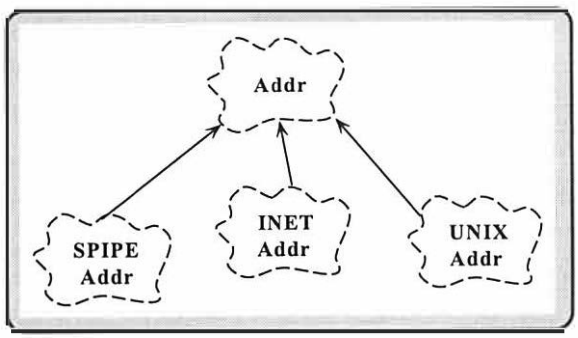


Figure 19: The SOCK_SAP Address Class Hierarchy

• **Inline performance critical methods:** To encourage developers to replace existing low-level network programming interfaces with C++ wrappers, the SOCK_SAP implementation must operate efficiently. To ensure this, methods in the critical performance path (such as the SOCK_Stream recv and send methods) are specified as C++ inline functions to eliminate run-time overhead. Inlining is both time and space efficient since these methods are very short (approximately 2 or 3 lines per method). The use of inlining implies that virtual functions should be used sparingly since most contemporary C++ compilers do not fully optimize away virtual function overhead.

• **Design auxiliary classes that shield applications from error-prone details:** *e.g.*, SOCK_SAP contains the Addr class hierarchy (shown in Figure 19). This hierarchy supports several diverse network addressing formats via a type-safe C++ interface. The Addr hierarchy eliminates several common programming errors (such as forgetting to zero-out a sockaddr addressing structure) associated with using the C-based family of struct sockaddr data structures directly.

• **Combine several operations to form a single operation:** *e.g.*, the SOCK_Acceptor is a factory for passive connection establishment. Its constructor performs the socket calls socket, bind, and listen required to create a passive-mode listener endpoint.

• **Supply default parameters for typical method argument values:** *e.g.*, the addressing parameters to accept are frequently NULL pointers. To simplify programming, these values are given as defaults in SOCK_Acceptor::accept so that programmers need not provide them.

6 Concluding Remarks

An important class of applications require high-performance streaming communication. Bandwidth-intensive and delay-sensitive streaming applications like medical imaging or teleconferencing are not supported efficiently by contemporary CORBA implementations due to data copying, demultiplexing, and memory management overhead. As shown in Section 2, this overhead is often masked on low-speed networks like Ethernet and Token Ring. On high-speed networks like ATM or FDDI, however, this overhead becomes a significant factor limiting communication performance.

The ACE socket wrappers described in this paper provide a high-performance network programming interface that shields developers from lower-level details of sockets or TLI without sacrificing performance. The ACE wrappers automate and simplify many aspects (such as initialization, addressing, and handling short-writes) of using lower-level network programming interfaces. They improve portability by shielding applications from platform-specific network programming interfaces. Wrapping sockets with C++ classes (rather than stand-alone C functions) makes it convenient to switch wholesale between different network programming interfaces by using parameterized types. In addition, as shown in Figure 2, the ACE socket wrappers do not introduce any significant overhead compared with programming with socket directly.

The primary drawback with the ACE network programming wrappers is that they do not address higher-level issues related to system reliability and availability, flexibility of object location and selection, support for transactions, security, and deferred process activation, and the exchange of binary data between different computer architectures. For example, programmers must provide explicit support for presentation layer conversions in conjunction with the ACE wrappers. Therefore, these wrappers are most useful when the datatypes simple, like those used by the high-performance streaming applications described in this paper.

The ACE C++ wrappers for sockets may be integrated with CORBA to enhance the performance of streaming applications. We've combined CORBA and the ACE wrappers in a high-speed teleradiology system that transfers 10-40 Mbyte medical images over ATM. In this system, CORBA is used as a signaling mechanism to identify endpoints of communication in a location-independent manner. The ACE wrappers are then used to establish point-to-point TCP connections and transmit bulk data efficiently across the connections. This strategy builds on the strengths of both CORBA and ACE.

ACE has been ported to many versions of UNIX and Windows NT and is currently being used in many commercial products including the Bellcore and Siemens Q.port ATM signaling software product, the Ericsson EOS family of telecommunication monitoring applications, the System Control Segment of the Motorola Iridium project, and a high-speed enterprise-wide medical image delivery system for Kodak Health Imaging Systems.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1.2 ed., 1993.

[2] J. Dilley, "OODCE: A C++ Framework for the OSF Distributed Computing Environment," in *Proceedings of the Winter Usenix Conference*, USENIX Association, January 1994.

[3] Microsoft Press, Redmond, WA, *Object Linking and Embedding Version 2 (OLE2) Programmer's Reference, Volumes 1 and 2*, 1993.

[4] S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[5] Sun Microsystems, *Network Interfaces Programmer's Guide*, Chapter 6 (TLI Interface) ed., 1992.

[6] D. C. Schmidt, "ASX: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[7] D. C. Schmidt, "Experience with a System of Reusable Design Patterns for Motorola Iridium Communication Software," in *Submitted to the Theory and Practice of Object Systems (special issue on Patterns and Pattern Languages)* (S. P. Berczuk, ed.), Wiley and Sons, 1995.

[8] G. Blaine, M. Boyd, and S. Crider, "Project Spectrum: Scalable Bandwidth for the BJC Health System," *HIMSS, Health Care Communications*, pp. 71–81, 1994.

[9] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[10] USNA, *TTCP: a test of TCP and UDP Performance*, Dec 1984.

[11] D. C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 489–506, May 1993.

[12] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 200–208, ACM, Sept. 1990.

[13] M. DoVan, L. Humphrey, G. Cox, and C. Ravin, "Initial Experience with Asynchronous Transfer Mode for Use in a Medical Imaging Network," *Journal of Digital Imaging*, vol. 8, pp. 43–48, February 1995.

[14] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the 2nd C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[15] S. W. O'Malley, T. A. Proebsting, and A. B. Montz, "Universal stub compiler," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, Aug. 1994.

[16] K. Birman and R. van Renesse, "RPC Considered Inadequate," in *Reliable Distributed Computing with the Isis Toolkit*, pp. 68–78, Los Alamitos: IEEE Computer Society Press, 1994.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

[18] D. C. Schmidt, "Design Patterns for Active and Passive Establishment of Network Connections," in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95*, (Aarhus, Denmark), August 1995.

[19] W. R. Stevens, *UNIX Network Programming*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[20] G. Booch, *Object Oriented Analysis and Design with Applications (2nd Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

Program Explorer: A Program Visualizer for C++

Danny B. Lange Yuichi Nakamura

IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato-shi
Kanagawa-ken 242, JAPAN
e-mail: danny@acm.org

Abstract

Despite the obvious advantages of using object-oriented (O-O) program visualizers in system understanding and debugging, they are still rarely found in the programmers's tool box. One reason for this that such visualizers often fail because of their inability to handle problems of a realistic scale. In our research, we have addressed the scalability problem by integrating static and dynamic program information to produce abstract and yet accurate views of complex O-O systems that often provide more useful information than can be obtained by reading the source code. This is the approach we followed in designing Program Explorer, a research prototype for C++ program visualization, which has been used to examine large O-O systems such as Stanford's Interviews library and Taligent's CommonPoint frameworks.

1 Introduction

Whether we want to re-use or debug an object-oriented (O-O) system, we must acquire a thorough understanding of the static and dynamic properties of the system. Although the use of object orientation has changed software development for the better, it has not exactly made programs easier to understand.

Inheritance, **polymorphism**, and **encapsulation** are all good O-O concepts, but also tend to make the actual designs in which they are used more difficult to comprehend. Inheritance makes it difficult to "read" the behavior of a particular object, since that object can belong to a chain of, say, five to ten classes; polymorphism often makes it difficult to determine which method is actually executed in a given object; and encapsulation makes it difficult to understand that no object is an island, but rather that each is a part of a cooperative network of objects.

In fact, understanding O-O systems is the art of combining knowledge about the **concrete** (objects and their interaction) and the **abstract** (classes and their relationships). One cannot fully understand an O-O system by simply considering its concrete or abstract properties in isolation. The former consist of the visible results of the execution of the system, while the latter consist of what we can expect from the system. Understanding evolves from a knowledge of the relations between these two sets of properties.

The process of understanding programs is notoriously difficult and cumbersome. Often many different classes are involved and interaction turns out to be non-trivial. The complexity of both the design and some O-O concepts frequently makes it difficult to verify the scenarios we construct with pen and paper.

Tools such as **CIA++** [9] and **GraphLog** [2] have been developed to help in understanding O-O programs. The problem, however, is that information given by this type of tool for real-

world programs is sometimes very difficult to comprehend. There are very few ways of distinguishing relevant information from less relevant information. The root of the problem is that these tools more or less display the obvious: an unfiltered graphical representation of the source code that is just as difficult to comprehend as the source code itself.

Our approach is to combine program execution with the understanding of objects and interactions, and the static program information (source code) with the understanding of classes and their relationships. This allows us to determine which classes are relevant and how program behavior changes according to our interaction with the running program.

The challenge of this approach is that dynamic analysis of O-O systems involves generating huge amounts of program information that is hard to digest, especially if the information is presented in a purely textual format. The situation is not unique. Other scientific areas characterized by huge amounts of data face the same problem. One solution that has become increasingly popular is called scientific visualization. It has in many cases proven to be a particularly good way of presenting a large amount of information [16].

Two O-O program visualization tools that utilize dynamic information are **Object Visualizer** [17, 18] and **HotWire** [12]. Both rely on visual effects to draw attention to program anomalies, rather than giving exact information. Although this approach appears efficient for detecting problems and, to some extent, for localizing them, it hardly provides the exactness that programmers need in order to understand a problem and correct it.

In our research, we have developed a mechanism capable of amplifying patterns of object interaction in program visualizations. Our approach is based on the observation that static information can leverage dynamic information and vice versa. The technique is to let the dynamic information impose (1) a sense of relevance on the static entities, such as which classes or member functions are actually important during a certain phase of execution, and (2) a sense of sequence, such as the order in which member functions are called. In contrast, static information allows us to focus on (1) objects of certain classes, and (2) properties related to certain classes.

Hence, the two main issues that have to be addressed in order to visualize O-O programs are the availability of program information (capturing static as well as dynamic information), and the scalability of information processing and presentation (that is, the ability to create useful visualizations of real-world systems). We have addressed both these issues in the development of a research prototype for program visualization named **Program Explorer.**

Program Explorer is a system for understanding C++ programs through visualization. Its purpose is to provide class- and object-centered views of the structure and behavior of large C++ systems, with information accurate enough to enable programmers to re-use and maintain undocumented parts of such systems. The tool should be graphically oriented and should provide interactive hypertext-like navigation of program entities in running programs.

In the following section, Program Explorer's way of coupling static and dynamic program information is described and some visualizations are displayed. Section 3 presents the **Program Database**, the source of static program information. Sections 4 and 5 describe how the dynamic program information is generated and retrieved. Section 6 presents our conclusions.

2 Information Coupling and Visualization

From the Program Database, Program Explorer can retrieve static program information for a number of interesting visualizations such as class inheritance hierarchy, function calls, and variable access. From the execution trace of a running system it can generate a number of vi-

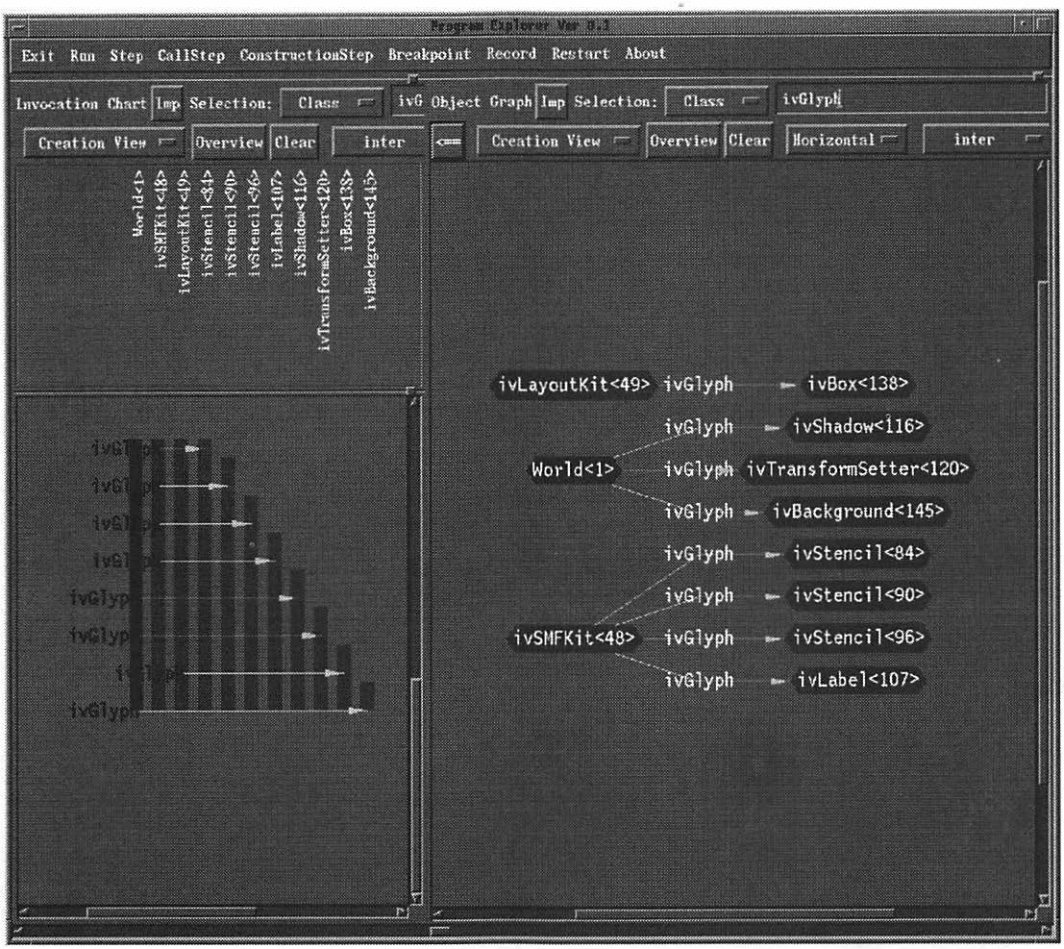


Figure 1: *Class-to-Object Coupling: Vertical Selection.*

sualizations that can give insights into its dynamic properties, such as object creational hierarchy, object calls, and object usage. It is thus possible to investigate the exact sequences of interaction among several objects needed to accomplish some task, and to detect program anomalies such as objects being created but never destroyed, and objects being invoked after they have been destroyed.

However, in our experience the main problem with pure static and pure dynamic models is their limited applicability to systems of a realistic size. The reason for this is that, in a sense, pure static views as well as pure execution views just show the obvious. That is, they show what is already present in the source code and in the computer memory during program execution. What helps in program understanding is the coupling of the abstract and the concrete, that is, of static and dynamic information. In Program Explorer we have found two coupling techniques that are particular useful, namely, **Classes-to-Objects** and **Objects-to-Classes**.

The Classes-to-Objects coupling can be used to filter dynamic information by means of static information. In Program Explorer, we have defined two particular categories of selection based on the class inheritance graph: **vertical** and **horizontal selection**. By vertical selection we mean the process of selecting objects of a given class. By horizontal selection we mean the se-

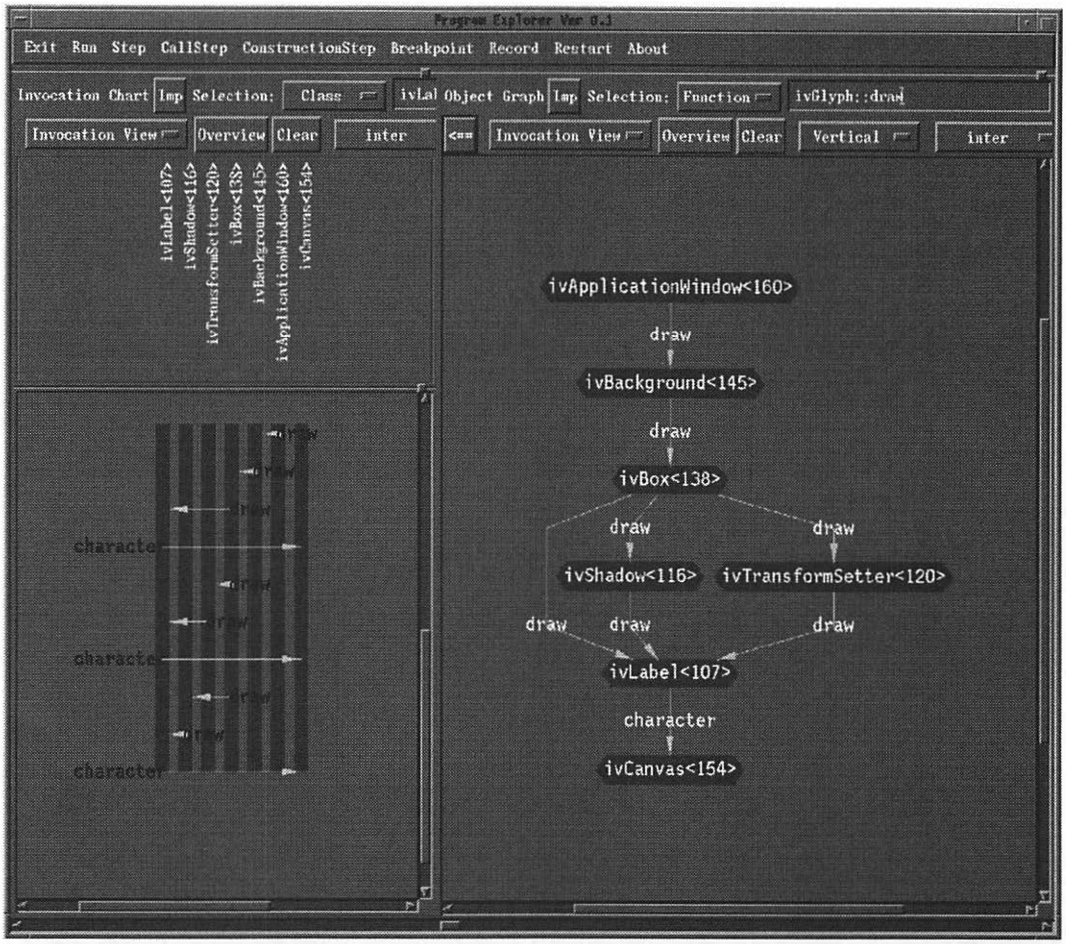


Figure 2: *Class-to-Object Coupling: Horizontal Selection.*

lection of certain object properties related to a specific class. Often, we use vertical selection to select a group of objects, and then use horizontal selection to study particular aspects of their interaction.

We have created an Interviews [14] sample program to examine the use of flyweight[1] objects in Interviews (see Appendix A and Figure 7). We know that `ivGlyph` is the root class for all graphical objects: thus, by retrieving objects derived from this class we can examine all graphical objects produced by this sample program. Figure 1 demonstrates such use of vertical selection. In the right pane we see the **Object Graph** of graphical objects and their creators. The nodes in the graph are objects identified by their class name and a unique number (`World<1>` represents the un-instrumented creators). The arcs represent creational relationships. In the left pane we find the **Invocation Chart**, which displays object longevity (the lengths of the bars) and the order of creation (from top to bottom).

Following the vertical selection, in Figure 2 we show an example of horizontal selection. The class `ivGlyph` defines a virtual function `draw` that is implemented in each of the derived classes. By selecting `ivGlyph::draw` we focus on a single behavioral aspect of these objects. The hierarchical drawing process from window

[1] Flyweight objects are fine-grained objects that can be shared efficiently.

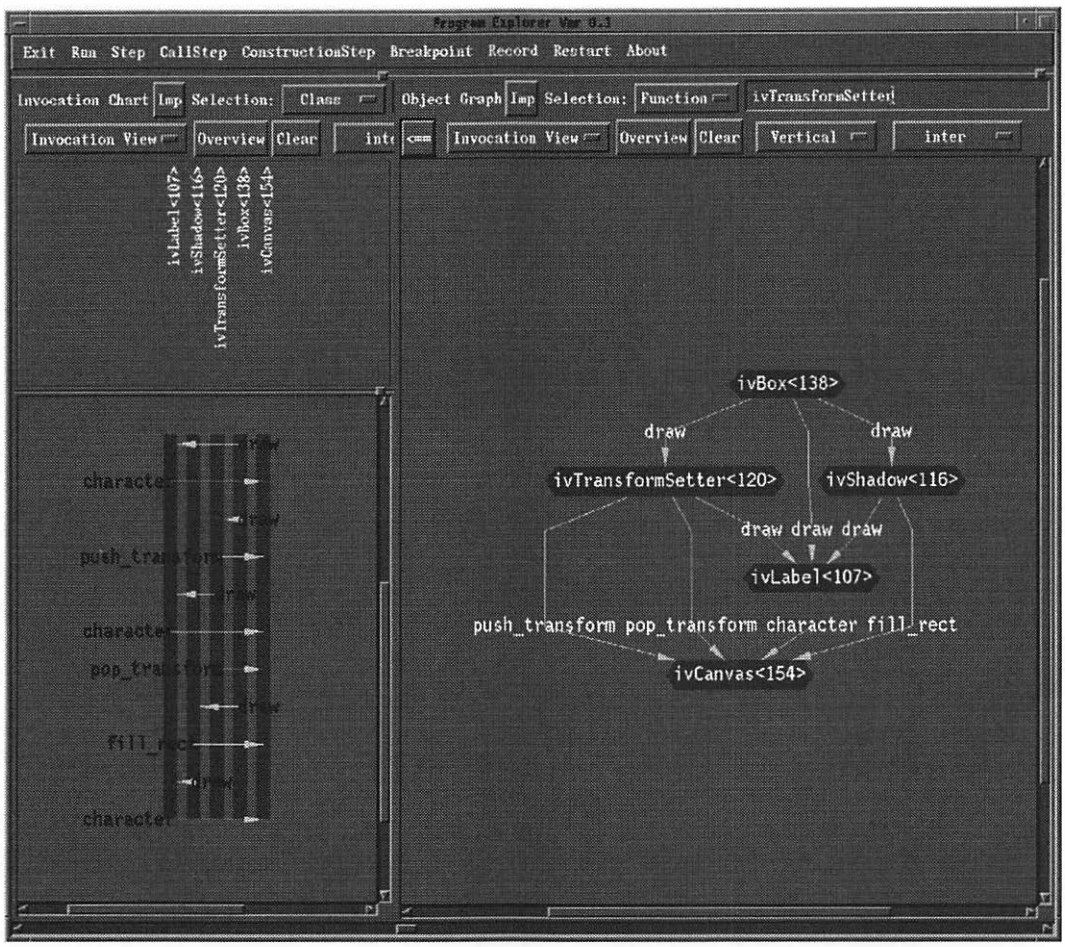


Figure 3: *Moving Focus to* `ivTransformSetter`.

to canvas becomes very clear, particularly in the invocation chart. It also shows how the label, as a flyweight object, is reused and drawn in three different contexts: plain, with a shadow, and transformed (rotated).

By Objects-to-Classes coupling we mean the transfer of dynamic information to the static domain. The purpose of this coupling is to filter huge amounts of static information in such a way that only the relationships actually used are in focus. Such visualizations can be used to express class-proximity (quantification of dynamic information to determine how dependent classes are on each other), or they can be used in class-based diagrams to describe object communication by numbered call relationships among classes, indicating the order in which invocations are supposed to take place. Surprisingly informative visualizations can be obtained by applying static and dynamic information coupling to large systems.

Orthogonal to the static and dynamic representations of O-O program executions are their **visual representations**. The purpose of a visual representation is to communicate the content of a given view of an O-O program and its execution. Different visual representations have different characteristics that make them more or less suited for particular views. In our research we have implemented and studied a number of interesting visual representations including graphs, bar charts, and matrices. The use of

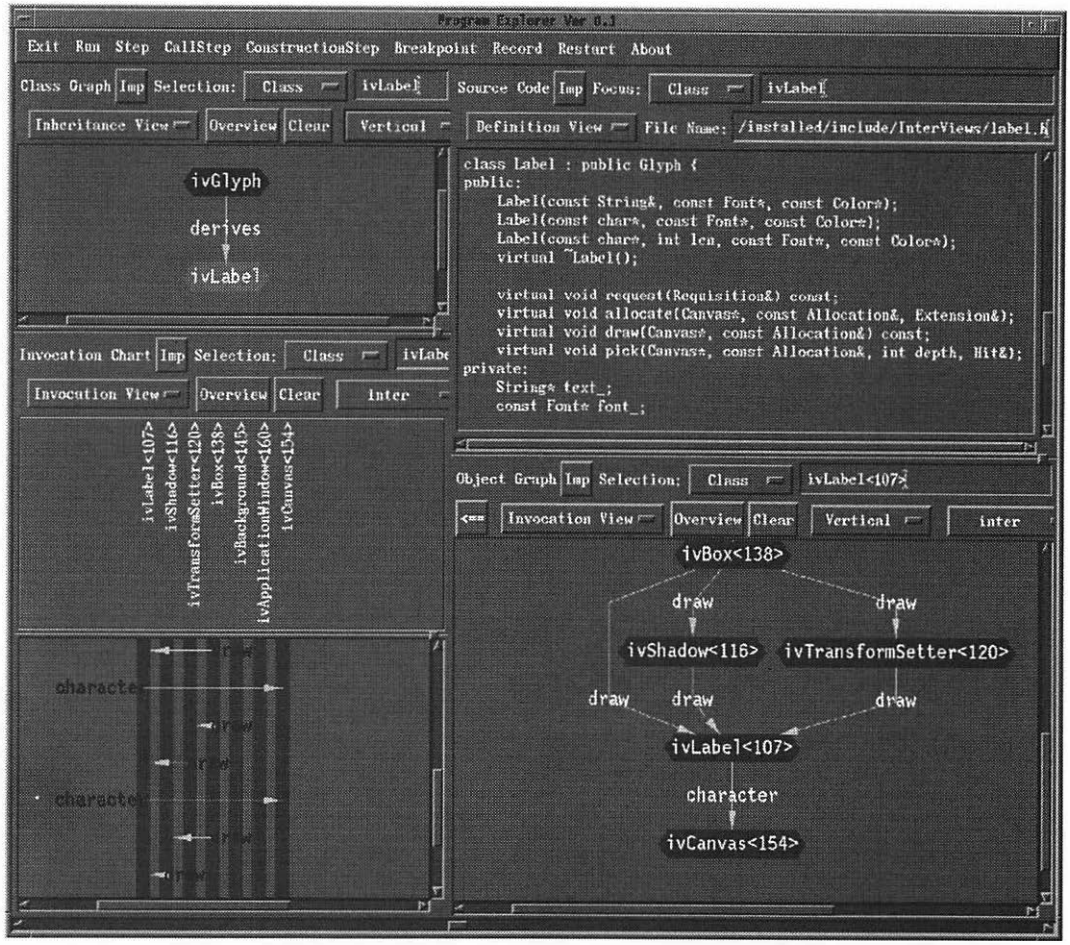


Figure 4: *View Integration: Four Views of* `ivLabel`.

color has also been explored. Our color convention (which can be re-defined by the user) is red for objects, green for classes, blue for free functions, and brown for the un-instrumented part of the system. Highlighting is used to display selected entities.

Two interaction techniques are used in Program Explorer's GUI to deal with the issue of scalability. **Navigation** is the technique used within a given view to explore it in an step-by-step fashion comparable to that of hypertext linking. In the previous example, we can move the focus to the `ivShadow` and `ivTransformSetter` objects to examine how a label is actually given a shadow and how it is rotated. In Figure 3 we have moved the focus to one of the above objects, and we can see how `ivTransformSetter` actually instructs the canvas to rotate the label object without the latter's knowledge. Navigation allows the user to focus on certain parts of a graph while ignoring others.

A focus can also be exported to another view. As is shown in Figure 4, Program Explorer's GUI consists of four panes, of which three are graphical and one is textual. The **integration** of these four panes through the focusing mechanism enables the user to view static properties in one pane and dynamic properties in another. We regard this as a GUI-based coupling of static and dynamic information.

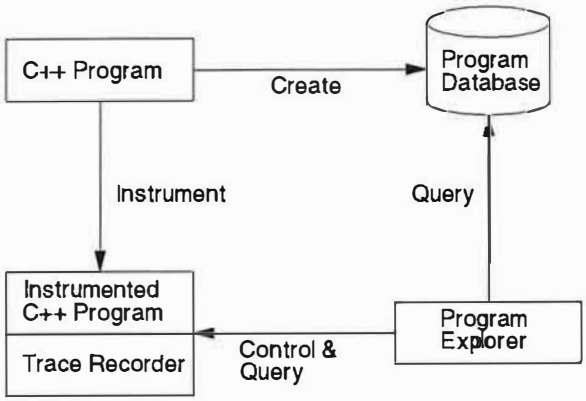


Figure 5: *The System Architecture of Program Explorer.*

3 Static Program Information

Retrieving static program information for C++ programs is strictly speaking not a part of Program Explorer. It is retrieved by the **Program Database** from so-called pdb-files generated by IBM's **xlC** compiler.

An overview of the system architecture of Program Explorer can be seen in Figure 5. The system includes a program database for C++, an **instrumentation utility** for augmenting C++ programs with code that produces trace information, a **Trace Recorder** linked to the instrumented programs that captures the trace, and finally, Program Explorer, which controls program execution and presents static and dynamic information through its GUI.

The Program Database is a stand-alone application that implements the schema of static program information from Table 1 and provides a full Prolog interface to clients. A simple query

```
pd_class(Cid, Cname, class,_,_)?
```

will return a set of pairs of (*ClassID*, *ClassName*). The query can be extended to

```
pd_inherit(DerID, BaseID,_,_,_,_),
pd_class(DerID, DerName, class,_,_),
pd_class(BaseID, baseName, class,_,_)?
```

which returns the set of binary inheritance relationships between classes. The built-in Prolog interpreter can also compute recursive queries, and the uniform representation of facts and rules in Prolog allows clients to append rules to the database and thus create their own logic programs as a part of the database.

4 Program Instrumentation

Collecting accurate information on a running C++ program is not easy. Without a meta-class concept, there are basically three ways of collecting such information: (1) extend the compiler so that it adds trace-generating code to the normal code, (2) use debugging techniques, or (3) augment the source code with trace-generating code.

Many compilers offer options for generating profiling information that can be used to detect "hot spots" in the code. Traces of this type do not provide enough information for our kind of visualization. Ideally, compilers can be modified to insert code that produces detailed trace information, but we considered that task to be beyond the scope of our work.

Another way of retrieving dynamic information is to use debugging techniques. Normally, this is done by setting breakpoints at the functions in a program. However, when dynamic information is collected in this way, a great deal of time is spent by the system in context switching between the process of the target program and the process of Program Explorer. Processes are heavyweight processes in AIX and a context switch from process to another is a costly operation.

The third solution is to augment the source code with trace-generating code. This technique is often termed **program instrumentation**. C++ preprocessors have been used to instrument programs [17], but a complete instrumentation would require a semantic analysis comparable to the one carried out in the compiler. We found instrumentation to be an

Table 1: *Schema for Static Information.*

- **Facts on entities:**

```
pd_directory(ID, Name, PathName, ParentDirID)
pd_file(ID, Name, PathName, Time, Language, DirID)
pd_class(ID, Name, ClassType, IsSOM, SOMName)
pd_function(ID, Name, PrototypeString, FuncStorageClass, Const, Inline, Overload,
            Operator, VirtualSpecifier, FuncMiscAttriute, Volatile, IsSOM, SOMName)
pd_variable(ID, Name, DeclarationString, VarStorageClass, Const, Volatile)
pd_enumeration_tag(ID, Name)
pd_enumerator(ID, Name, EnumID)
pd_macro(ID, Name, NumberOfArguments)
pd_typedef(ID, Name, DeclarationString)
pd_label(ID, Name)
pd_template(ID, Name, TemplateType, LongName)
```

- **Facts on relationships:**

```
pd_defined(ID, ScopeID, FileID, Line, Column)
pd_declared(ID, ScopeID, FileID, Line, Column)
pd_used(ID, ScopeID, FileID, Line, Column)
pd_used_implicit(ID, ScopeID, FileID, Line, Column)
pd_used_lvalue(ID, ScopeID, FileID, Line, Column)
pd_member(ClassID, MemberID, AccessSpecifier, Offset)
pd_friend(ClassID, FriendID, Line, Column, FileID)
pd_inherit(DerivedClassID, BaseClassID, AccessSpecifier, Virtual, Order, FileID)
pd_include(FileID, IncludedFileID, Line)
pd_instantiated(InstantiatedID, TemplateID)
pd_source2pdb(SourceFileID, PdbFileID, CompilerOption, Language)
pd_call(CallerID, CalleeID)
```

- **Facts on types of names:**

```
pd_typeof_variable(ID, Type)
pd_typeof_function(ID, ReturnType, NumberOfArguments, ListOfArgumentTypes)
pd_typeof_typedef(ID, Type)
```

acceptable solution, but instead of using a C++ preprocessor we decided to rely on the Program Database for accurate instrumentation information. The contents of the Program Database are compiler-generated and thus very exact with respect to the semantics of program entities and their physical location.

Instrumentation. Program Explorer provides selective instrumentation on a class-wise basis. The GUI of Program Explorer allows the user to specify which classes should be instrumented. A directory-file-class hierarchy allows flexible selection of whole directories, files, and classes. This technique is suitable for avoiding trace information from highly active classes and from classes that are trivial or already well understood. We intend to extend this technique to encompass functions as well as variables.

Our aim has been to instrument C++ programs to produce complete and accurate trace information. For that purpose it is necessary to capture events related to **object longevity**, **function invocation**, and **variable access**. Program events are captured by the Trace Recorder through the internal protocol (see Figure 6 and Table 2). The Trace Recorder processes events to produce a trace, which it also stores. Program execution is controlled and trace information is queried by Program Explorer through the Trace Recorder's external interface.

It should be emphasized that the instrumentation code shown below is in no way specific to IBM's `xlC` compiler. The code fragments are generally applicable for tracing C++ programs, and only the way in which we insert those frag-

Table 2: *Internal Protocol for the Trace Recorder.*

	Command	Arguments			
Longevity	`Allocate`	*ObjectID*	*ClassID*	*MemoryAddr*	
	`Deallocate`	*ObjectID*			
Invocation	`Construct`	*ObjectID*	*ClassID*	*FunctionID*	
	`Destruct`	*ObjectID*	*ClassID*	*FunctionID*	
	`Enter`	*ObjectID*	*ClassID*	*FunctionID*	
	`Leave`	*ObjectID*	*ClassID*	*FunctionID*	
Access	`Usage`	*ObjectID*	*VariableID*	*RetrFunc*	*ObjectAddr*

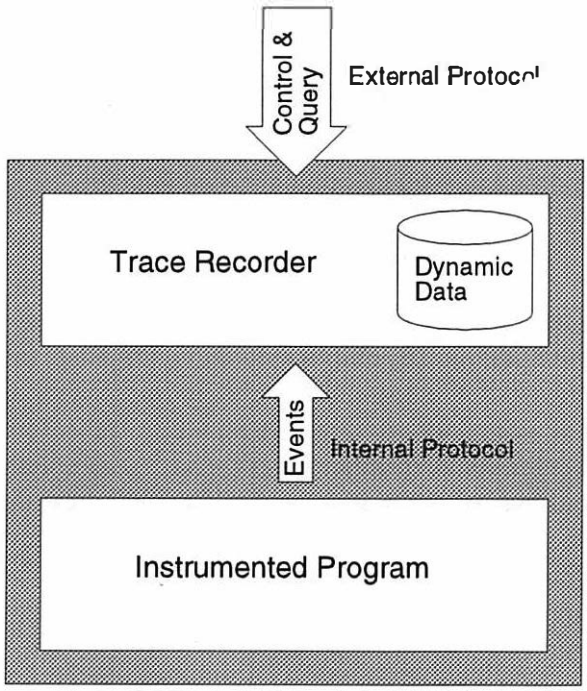


Figure 6: *The Executable: Program and Trace Recorder.*

ments is specific to the `xlC` compiler and the Program Database.

Object Longevity. Two events related to object longevity are captured: creation and destruction of an object. For this purpose every instrumented class is equipped with a "identity object", `_PEoid`:

```
class A {
   PE_Oid _PEoid;
...
}
```

For this instrumentation we use C++'s automatic construction and destruction of class members. The constructor of `PE_Oid` assigns a unique number to the object and calls `Allocate` in the Trace Recorder, whereas the `PE_Oid`'s destructor calls `Deallocate`. The constructor and destructor functions of `A` are not suitable for capturing object creation and destruction, since such events actually take place respectively before and after these functions are called.

Where should we put `_PEoid` in the case of inheritance? We put it into each instrumented class, even if these classes appear in the same inheritance path. To avoid allowing one object to have multiple identities, subclass `_PEoids` "inherit" their unique identity (an integer value) from the base class. In the case of multiple inheritance this mechanism works the opposite way.

```
class B : public A {
   PE_Oid _PEoid(A::_PEoid.oid());
...
}
```

Why not use the value of `this` as a unique id of an object? This would not work, for several reasons. First, if multiple inheritance, possibly with virtual base classes, is involved, different

parts of an object (depending on which class in the inheritance path that part is representing) will return different values of `this`. Second, when an object is deallocated and a new object is created in the same space, it would be difficult to maintain a unique trace as object ids would no longer be unique over time.

Function Invocation. Function invocations are captured with a mechanism identical to the one described for object longevity. `_PEtmp` is a local class variable that is automatically constructed when the function is invoked, and destructed upon its return:

```
A::F() {
    PE_Func _PEtmp(_PEoid.oid(),
                   ClassID,
                   FunctionID);
...
}
```

The constructor of `PE_func` calls `Enter` in the Trace Recorder, and the destructor calls `leave`. One of the benefits of this approach is that only the callee is instrumented, and since one function may have many callers, it more efficient than instrumenting call locations.

The C++ compiler silently generates a number of functions if they are not explicitly defined by the programmer. Such functions are the *constructor*, *destructor*, *copy-constructor*, and *assignment operator*. For more details on this subject, consult the C++ ARM [5].

Regardless of the implicitness of these functions, they often play an important role in the execution of an O-O program and thus cannot be ignored. The copy-constructor creates new objects, and along with the assignment operator it copies the state of one object to another. No source code exists for compiler-generated functions, so we need to make these functions explicit.

Below is an example of a constructor and destructor. Notice that `PE_Constr` and `PE_Destr` are used instead of `PE_Func` to distinguish these three types of function invocation. The constructor of `PE_Constr` calls `Construct` in the Trace Recorder and `PE_Destr` calls `Destruct`. The constructor instrumentation is as follows:

```
A::A() {
    PE_Constr _PEtmp(_PEoid.oid(),
                     ClassID,
                     FunctionID);
...
}
```

and the destructor instrumentation is as follows:

```
A::~A() {
    PE_Destr _PEtmp(_PEoid.oid(),
                    ClassID,
                    FunctionID);
...
}
```

The copy-constructor and the assignment operator are more complicated to create than the above constructor and destructor. In both cases it is necessary to create separate initialization lists for scalar member variables and assignment routines for array members. Below is an example of an explicit copy-constructor. Notice that `_PEoid()` in the initializer list ensures that the copied object receives a new identity different from the originator (`rhs`).

```
A::A(A& rhs) : _PEoid(),
               r(rhs.r), s(rhs.s),...
{
    PE_Constr _PEtmp(_PEoid.oid(),
                     ClassID,
                     FunctionID);
    Array member assignment
}
```

Free functions (non-member functions) are also instrumented. In their case the class identity and object identity are set to zero.

Variable Access. Variable access is more difficult to capture than function invocations. The

Table 3: *Schema for Dynamic Information.*

- **Object longevity:**

```
create(SrcObjID, SrcFunID, TgtObjID, ClassID, Time)
destroy(SrcObjID, SrcFunID, TgtObjID, Time)
```

- **Interactions:**

```
invoke(SrcObjID, SrcFunID, TgtObjID, TgtFunID, Time)
access(SrcObjID, SrcFunID, TgtObjID, VariableID, Time)
```

- **State:**

```
value(VariableID, Value, Time)
```

approach we have taken is to "functionalize" variable access. That is, each definition of a member variable is attributed by an access function to be used instead of direct variable access. The original member definition

```
B* b;
```

is supplemented by two functions. The first serves to capture variable access:

```
B*& b_PE() {
    PE_usage(_PEoid.oid(),
             VariableID,
             A::b_PEvalue);
    return b;
}
```

while the second is used to retrieve the value of the variable:

```
static void* b_PEvalue(void* o) {
    return (void*)((A*)o)->b;
}
```

The access function (`PE_usage(...)`) notifies the Trace Recorder about variable uses. It also forwards a pointer to a function able to return the value of the variable. The reason the Trace Recorder does not receive the value directly is that variable assignment first takes place after the access function has returned. Member variable access is modified by adding the function-suffix `_PE()`. This suffix works for both reading a variable:

`b->foo();` *becomes* `b_PE()->foo();`

and writing to a variable:

`b = new B;` *becomes* `b_PE() = new B;`

5 Trace Recording and Execution Control

When an instrumented program has been compiled and linked with the Trace Recorder, it is ready to be executed by Program Explorer. Below we describe trace recording and how it is controlled and used by Program Explorer.

Trace Recording. The atomic events of a running program are captured by the Trace Recorder and transformed into a sequence of binary relations. The event in which an object is created is transformed into a relation that specifies the identity of the object being created as well as that of its creator. Function invocations are captured as a sequence of `Enter`- and `Leave`-events that can easily be converted into the corresponding series of binary call relationships. Binary relations are more convenient than raw events with regard to storage management and query processing. The schema for representing these relationships is given in Table 3.

Since the Trace Recorder stores and manages the trace, Program Explorer has to pose queries

Table 4: *External Protocol for the Trace Recorder.*

	Command	Input	Output
Execution Control	`exit`		
	`run`		*Focus Record*
	`step`		*Focus Record*
	`callStep`		*Focus Record*
	`returnStep`		*Focus Record*
	`constructionStep`		*Focus Record*
	`usageStep`		*Focus Record*
	`setBreakPoint`	*Focus Record*	
Trace Recording	`invocRecording`		
	`usageRecording`		
Query Processing	`about`		*Result*
	`getInvocations`	*Focus Record*	*Result*
	`getConstructions`	*Focus Record*	*Result*
	`getUsages`	*Focus Record*	*Result*
	`getPointers`	*Focus Record*	*Result*

to the Trace Recorder in order to produce visualizations of dynamic information. The query interface to the Trace Recorder is a part of its external protocol given in Table 4. Unlike the Program Database, the Trace Recorder only provides a fixed number of query functions. This restriction has been caused by a requirement of low response times and space-efficient storage of large traces. Still, the query mechanism is very flexible, since each query can take a **Focus Record** as argument. A Focus Record specifies any meaningful selection of a static or dynamic program entity or relationship, such as *class*, *object*, *function*, or *invocation*.

Queries take the form of

`someQuery(`*ClassID*, *SrcObjID*, *TgtObjID*, *FuncID*/*VarID*)

and return lists of

`Result(`*SrcClassID*, *SrcObjID*, *TgtClassID*, *TgtObjID*, *FuncID*/*VarID*)

Notice that, since no text information is exchanged between Program Explorer and the Trace Recorder, the Program Database acts as a name server. Using unique class, function, and variable identifiers gives a very compact trace, lowers the communication overhead, and allows the system to distinguish overloaded names.

Execution Control. The instrumented program is executed under the control of Program Explorer. That is, Program Explorer uses the control interface of the Trace Recorder's external protocol to run the instrumented program or to execute it in a single-step mode. This mechanism is well known from debuggers and very suitable for localization (finding a spot of particular interest). Whenever execution in the instrumented program is halted (that is, when the program exits, or reaches a breakpoint, or when a signal from the operating system is received), a Focus Record is returned to Program Explorer, allowing it to retrieve information about the events that led to that halt. Unlike in debuggers, this information includes full details of all the recorded events up to the halt, and not only the contents of the call stack.

The Trace Recorder also allows selective trace recording. Both invocation and variable usage recording can be switched on and off independently, thus limiting the generation of trace information. This technique and the selective instrumentation mentioned in Section 4 are the two means of reducing trace generation. Selective instrumentation is limited to compile-time. In the future we would like to extend this mechanism to run-time, so that selective trace recording is supported by two orthogonal concepts: program-entity-based and breakpoint-based selection.

Statistics. Table 5 shows statistics from the trace recording of an instrumented version of the Interviews library. For some common Interviews sample and application programs we show the number of different class and function definitions involved in a particular execution, and the number of actual object creations and function invocations. In these examples, no attempts were made to limit trace generation. An important observation that can be drawn from this table, is that, while the amount of dynamic information grows rapidly in proportion to the size and complexity of the application program, the size of the static information space remains almost constant. This observation supports our approach of using static information to leverage dynamic information.

6 Related Work

Two query-based program visualization tools — **CIA++** [9], developed at AT&T Bell Laboratories, and **GraphLog** [3], developed at the University of Toronto — focus on the static properties of O-O systems. CIA++ builds a relational database of information extracted from C++ programs. The database serves as a foundation for static analysis tools for displaying various views of the program structure. GraphLog is a visual tool for databases. Queries are posed by drawing graph patterns with a graphical editor. GraphLog has been used for visualizing and querying software structures [2]. Since both tools are limited to static program information, their visualizations are of limited interest unless very interesting[2] queries are posed. In our experience, however, writing such queries is often difficult and distracts the user's attention from the original goal of understanding a program.

Object Visualizer [17, 18] and **HotWire** [12], both developed at the IBM T. J. Watson Research Center, are dynamic O-O program analyzers that primarily rely on visual effects to draw attention to program anomalies rather than giving exact information. Both tools are based on the same program instrumentation mechanism for gathering execution information. This mechanism is seemingly less accurate than Program Explorer's, and does not generate information on implicit[3] functions, variable usages, and variable values. The execution model of Object Visualizer [18] is accumulative, and is intended for O-O profiling uses such as finding "hot spots" in classes and objects. Another capability of this tool is visualizing "memory leaks," that is, objects that are not deleted after use. HotWire is a visual debugger for C++ that allows the user to write custom visualizations in a scripting language. While this scripting mechanism seems to be very useful for algorithm and object animation, we fear that it distracts the user's attention from program debugging (which often is performed under strong time pressure). HotWire resembles Program Explorer more closely than does Object Visualizer. HotWire and Program Explorer support microscopic views into the state and behavior of individual objects, whereas Object Visualizer focuses on the overall picture characteristic of program profiling tools.

To our knowledge, the systems described in the above have been mainly applied to programs written in C++. However, visualizers have also

[2] Such as whether a base class directly or indirectly relies on properties of one of its derived classes (visualizing bad programming style).

[3] A number of C++ member functions are silently generated by the compiler.

Table 5: *Trace Recording Statistics.*

Application Program	himom	flyweight	preview	idemo	doc
LOC	29	31	197	635	15,210
Classes	42	53	84	112	85
Functions	263	365	514	731	564
Objects	152	271	9,893	9,437	17,404
Invocations	603	1,146	32,327	33,291	64,553

been constructed for other O-O languages such as Smalltalk (e.g., message diagraming [4], the TRICK system [1], and Portia [8]), and LISP dialects (e.g., GraphTrace [11]). A common feature of these systems is that they benefit from the openness of interpreted O-O languages. Objects actually exist at runtime in these systems, whereas runtime structures in C++ are flat and not very O-O.

7 Conclusion

Scalability has been the major issue in the design and implementation of Program Explorer. The issue is complex, since it concerns human as well as computer resources. However, we have in our research found a common denominator for addressing this issue: static-dynamic information coupling.

Let us take the computer resources first. With a practically complete instrumentation utility, Program Explorer has been successfully used to instrument and generate trace information for large O-O systems such as Stanford's Interviews library and some of Taligent's CommonPoint frameworks [15]. To reduce the amount of generated dynamic information, we rely on the coupling of static and dynamic information to perform selective class-wise instrumentation combined with the use of breakpoints to switch execution tracing on and off.

Human resources, on the other hand, are primarily related to the reduction of the cognitive load. For this, we also rely on the coupling of static and dynamic information to produce visualizations that combine dynamic properties with static properties and vice versa. Such filtered views allow the programmer to focus on certain aspects of system behavior while ignoring others. Finally, the GUI of Program Explorer relies on static-dynamic information coupling to produce hypertext-style integration between the different visualizations.

At the time of writing, three different prototypes of Program Explorer have been developed. In addition to the one described in this paper, we have developed a version based on debugging technology, which makes both program instrumentation and program database obsolete. The advantage of this system is a shorter edit-compile-explore turn-around time for the developers. However, the price paid is a greatly increased execution time. The third version, for IBM's System Object Model (SOM) [10], replaces program instrumentation with an extension to the SOM metaclass that captures method invocations [6] and uses the SOM repository for static information.

Currently, we are investigating the concept of **O-O Design Patterns** [7] with the goal of automating the processes of searching for and visualizing recurring designs in O-O systems. Viewing an O-O system from the perspective of design patterns often makes the detailed design more comprehensible. Our initial experience is that the static-dynamic coupling mechanisms described in this paper are very useful for pattern analysis [13]. The reason for this is that design patterns very often rely on "abstract behavior" defined in abstract classes but realized in concrete classes. We have found the vertical

and horizontal selection technique described in Section 2 to be particularly useful for showing design patterns. If we can formally express the semantics of design patterns, we have the basic means for realizing automated search and visualization.

Even though Program Explorer is not intended for debugging, it demonstrates a clear potential for visual debugging. Its class- and object-centered visual representations of static and dynamic program information are an ideal communication medium for programmers. Moreover, its ability to keep a history of function invocations, as well as accesses to variables and the values of those variables, suggest the possibility of a more efficient debugging process, where programmers are able to investigate the events that lead to a run-time error instead of just being told where the error occurred.

Acknowledgements

We wish to thank our many colleagues at the IBM Tokyo Research Laboratory for their contributions to the Program Explorer project, and in particular Dr. T. Kamimura for his unflagging support. We are also grateful to R. Thornton and R. Pfeiffer of Taligent for their kind assistance in testing Program Explorer on Taligent's CommonPoint frameworks, and to M. McDonald of IBM Japan for checking the wording of this paper.

References

[1] H. Böcker and J. Herczeg. What Tracers Are Made of. In *OOPSLA '90, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 89–99, 1990.

[2] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, 1992.

[3] M. Consens and A. Mendelzon. Hy$^+$: A Hygraph-based Query and Visualization System. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD Record, 22(2), pages 511–516, 1993.

[4] W. Cunningham and K. Beck. A Diagram for Object-Oriented Programs. In *OOPSLA '86, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 361–367, 1986.

[5] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[6] I. R. Forman, S. Danforth, and H. Madduri. Composition of Before/After Metaclasses in SOM. In *OOPSLA '94, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 427–439, 1994.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture.* Addison-Wesley, 1994.

[8] E. Gold and M. B. Rosson. Portia: An Instance-Centered Environment for Smalltalk. In *OOPSLA '91, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 62–74, 1991.

[9] J. E. Grass. Object-Oriented Design Archaeology with CIA++. *Computing Systems*, 5(1), pages 5–67, 1992.

[10] IBM. *SOMObjects Developer ToolKit, Users Guide.* IBM Corp., 1993.

[11] M. F. Kleyn and P. C. Gingrich. GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views. In *OOPSLA '88, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–205, 1988.

[12] C. Laffra and A. Malhotra. HotWire – A Visual Debugger for C++. In *Proceedings of USENIX C++ Technical Conference.* USENIX Association, pages 109–122.

[13] D. B. Lange and Y. Nakamura. Interactive Visualization of Design Patterns Can Help

in Framework Understanding. To appear in *OOPSLA '95. Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1995.

[14] M. A. Linton, P. .R. Calder, J. A. Interrante, S. Tank, and J. M. Vlissides. *InterViews Reference Manual Version 3.1*. The Board of Trustees of the Leland Stanford Junior University, 1992.

[15] W. Myers. Taligent's CommonPoint: The Promise of Objects. *IEEE Computer*, 28(3), pages 78–83, 1995.

[16] G. M. Nielson, B. D. Shriver, and J. Rosenblum. *Visualization in Scientific Computing*. IEEE Computer Society Press, 1990.

[17] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the Behavior of Object-Oriented Systems. In *OOPSLA '93. Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 326–337, 1993.

[18] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling Object-Oriented Program Execution. In *Proceedings of the 8th European Conference, ECOOP '94*. Lecture Notes in Computer Science, Vol. 821, pages 163–182, 1994.

A The Flyweight Sample Program

The program below starts with the creation of a session, a widget kit, and a layout kit. The widget kit is used to create a text label. A transformer is created and set to a 90 degrees rotation. In the session window, the label appears three times: normally, transformed (90 degrees rotation), and with a shadow background (see Figure 7).

```
int main()
{
   Session* session = new Session();
   WidgetKit& kit = *WidgetKit::instance();
   LayoutKit& layout = *LayoutKit::instance();
   Glyph* label = kit.label("HELLO");
   Transformer t; t.rotate(90.0);
   session->run_window(
      new ApplicationWindow(
         new Background(
            layout.hbox(
               label,
               new TransformSetter(label, t),
               new Shadow(label, 0, 0,
                          new Color(0.7, 0.7,
                                    0.7, 1.0)
                          )
            ),
            kit.background()
         )
      )
   );
}
```

Figure 7: *The Flyweight GUI.*

Software Configuration Management in an Object Oriented Database

Mick Jordan
mjj@Eng.Sun.COM

Michael L. Van De Vanter
mlvdv@Eng.Sun.COM

Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043 USA

Abstract

The task of configuration management for software development environments is not well supported by conventional files, directories, and ad hoc persistence mechanisms. Typed, immutable objects combined with ubiquitous versioning provide a much more sound basis. A prototype configuration management system currently under development relies upon persistent objects provided by a commercial object-oriented database system. Mechanisms and policies developed for this prototype simplify programming with persistent objects; results include simplicity of both design and implementation, as well as flexibility, and extensibility. Initial measurements suggest that performance is acceptable.

1 Introduction

A practical software system consists of hundreds or thousands of separate but related components. A software development environment (SDE) must help manage this collection by providing *configuration management* services. Closely related is *version management*, often associated with tracking the evolution of individual components. However, configurations naturally exist as versions, and it is the effective management of such configurations that distinguishes a truly useful SDE for large, multi-user projects.

An SDE typically relies on the persistent storage facilities provided by the underlying operating system, usually in the form of a file system with two basic mechanisms: files and directories. Unfortunately the weak functionality of these primitives does not permit adequate modelling of configurations. This has led to the suggestion that an SDE rely instead on a database management system (DBMS) [1].

Experience with SDEs built using traditional DBMS technology has not been encouraging. Indeed such systems have acquired a reputation for being resource hungry, and they have not found their way into widespread use. However, in recent years there has been a surge of interest and development in object-oriented database (OODB) technology. While the relational database community might argue that OODBs are fated to repeat the perceived failure of the network databases of the 1970's, there is no doubt that the better impedance match with everyday programming has a great appeal to many developers. Indeed one can argue that OODBs are becoming successful because they are delivering, in part, the promise of persistent programming languages [2].

This paper describes our experience building a prototype configuration management system using ObjectStore® [3], a commercial database system produced by Object Design Inc.®

We begin by reviewing the advantages of an OODB as infrastructure for an SDE and then provide an overview of the essential facilities provided by ObjectStore. Next we briefly describe the structure and operation of our configuration management system and explain how objects help simplify both design and implementation. We then turn to the mechanisms and policies that we have developed to exploit ObjectStore successfully in our prototype, many of which have wide applicability. After presenting some preliminary performance measurements we describe related work and discuss our conclusions and future plans.

2 Why a Database?

Traditionally an SDE is built on top of a file system. The inertia caused by existing file-based tools makes it quite difficult to change this situation, and any database solution must provide a way to integrate legacy tools. Meanwhile progress in tools continues, and the limits of file-based systems are being pushed. In particular, it is increasingly common to find ad hoc "databases" being added to file based SDEs. Examples include data to support browsing, C++ [4] template instantiation, and build state for *make* [5]. Managing these ad hoc persistent stores, particularly in the face of concurrent access, is difficult and error prone. Yet, there is no shortage of

object-based systems offering solutions, for example CORBA [6] and Microsoft OLE 2 [7]. However, although these systems provide varying degrees of support for persistent data, they currently do not scale to the task of supporting an SDE efficiently. In contrast, OODBs have matured to the point that we believe it is practical to deploy them as an alternative to file systems and ad hoc persistence mechanisms.

An OODB offers two main advantages over a file system as infrastructure for an SDE:

- A transactional store, and
- Precise modelling of application data.

2.1 Transactional Store

All updates to a database take place inside a transaction; they either complete in their entirety or not at all. Therefore the evolution of data in a database proceeds through a sequence of well-defined states. Furthermore, data modified in a transaction by one process is not visible to other processes until the transaction commits. Providing such a guarantee for a file-based system is substantially harder.[1] While the transactional property might be seen as a minor feature in a single user programming environment, it is extremely important in large, multi-user projects. Even in single user environments, the use of multiple processes can easily put data into an inconsistent state, for example should a process abort at an inopportune moment.

2.2 Precise Data Modelling

We believe that the capability to design objects that closely model the application domain is the more significant benefit of an OODB. File based systems are hampered by the performance and modelling discontinuities that occur at the file and directory boundaries. An OODB can provide a more consistent solution. In particular, there are substantial advantages to be gained from utilizing *fine-grain* objects: objects too small to be stored efficiently as individual files. That all objects are also *typed*, significantly improves the readability and reliability of programs that manipulate them. Traditional solutions to storing fine-grain objects in a file system involve *pickling* schemes [8] [9], whereby programming language objects are preserved in some way inside one or more files. Although this can work well for simple structures, it rapidly becomes unwieldy for the complex, linked structures that prevail in compilers and associated tools [10].

1. A transactional file system would provide such a guarantee.

3 ObjectStore

ObjectStore is a commercially available OODB that runs under UNIX® and Microsoft® Windows. It is primarily targeted to the C++ language, but support for Smalltalk [11] has been added recently. Our experience is with the C++ system for the Solaris® variant of UNIX®.

The impedance match between ObjectStore and C++ is generally very good. Objects are allocated in a database by overloaded variants of the C++ `operator new`. In consequence all objects in the database can be referenced by C++ pointers. ObjectStore transparently maps portions of the database into the address space of the C++ process, using virtual memory mapping primitives of the underlying operating system. Access to objects in the database must occur inside a transaction, and any pointers that are mapped within that transaction are not valid after it terminates. Multiple processes may access the same database, and ObjectStore enforces a multiple reader-single writer locking policy.

Within these constraints, and inside a transaction, manipulation of database objects is no different from manipulating virtual memory (heap) objects, and it is this feature that makes use of the database transparent. However, to avoid *lock conflicts* with other processes it is necessary to limit the time spent in each transaction. A lock conflict occurs when one process holds a write lock and another process requests a read or write lock, or vice versa; this delays the requesting process until the lock is relinquished. Since locks are acquired only as data is accessed, *deadlock* can occur when access patterns produce cyclic lock dependencies. In case of deadlock one transaction is automatically aborted and subsequently retried up to a user-defined maximum number of times.

In order to retain access to an object between transactions, it is necessary to use an ObjectStore *reference*: a kind of heavyweight pointer. Unless some care is used to minimize the use of references, much of the transparency is lost. We will discuss this issue in more detail in section 5.4.

ObjectStore is based on a client-server model. Databases are managed by a server process on a designated server machine. Client processes, typically on separate machines, communicate with the server to read and write data. Client-side caching is heavily used to minimize communication overhead.

A process can access multiple ObjectStore databases within a transaction, and database objects can contain references to objects in other databases. An update transaction that modifies data in multiple databases managed by different servers is implemented by a two-

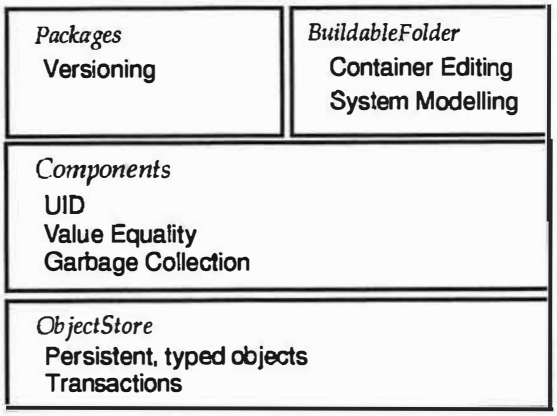


Figure 1. Architectural Layers in Forest

phase commit protocol.

We currently do not use ObjectStore's other features, such as querying, collections library, and generic version management. Our primary interest is in exploiting the shared, distributed, and transacted virtual memory model provided by the basic infrastructure.

4 Configuration Management in Forest

Our configuration management system is part of a larger project, Forest, that aims to provide an integrated, multi-user, software development environment for medium to large scale systems programming. A major goal of Forest is to discover whether current OODB technology is up to the task.

The goal of the configuration management system is to support the precise specification of a software system that may occur in many versions and variants. It must be capable of dealing with a spectrum of object types, from very small objects that might represent a fragment of a source program through to structured objects that might represent a complete system build containing thousands of objects. Intermediate in this spectrum are objects that represent object modules, programs, etc. In file-based systems the structure of these kinds of objects is typically defined by a *file format*. In object-based systems we expect such objects to be defined increasingly as abstract object types.

Figure 1 suggests how services are provided by different architectural layers of the prototype.

4.1 Components

Since the term object is rather overloaded, we use the term *component* to denote the objects that are manipulated by the Forest configuration management system. Although ObjectStore can store instances of any C++ type, we restrict components to be a subclass of an abstract C++ class `Component`, which defines a common set of operations and attributes.

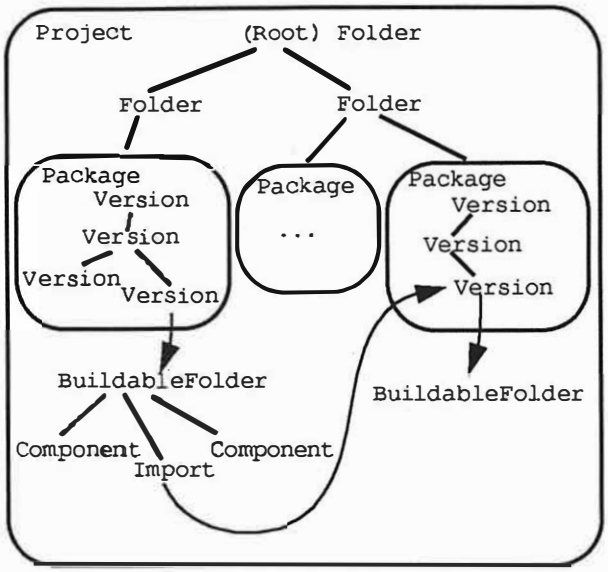


Figure 2. Abstract Example of a Forest Project

It is possible to construct arbitrarily complex collections of components, connected in an OODB, much more easily than one can with a file system. In practice, we enforce some familiar high level structure, partly to meet the goals of the configuration management system and partly for administrative and engineering reasons.

4.2 Projects

Components are allocated in and belong for their lifetime to a *project*. A project physically groups related components and, as the name implies, is intended to be shared by a group of collaborating programmers. A project is similar to a disk volume or mounted file system and is implemented as a single ObjectStore database.[1] Components in one project may refer to those in another, but only through special components called *links*.

At the outer level a project looks very like a conventional file system. A root *folder* (analogous to a file system directory) can contain other folders or arbitrary components. Two particular component types, *Package* and *BuildableFolder*, are the essence of the configuration management system; we will now describe these in more detail. Figure 2 suggests how these are related in practice.

1. We use the terms "project" and "database" interchangeably.

4.3 Packages and BuildableFolders

Forest follows the Vesta [12] approach to configuration management. A project contains a collection of heterogeneous *source* components, organized into families called *packages*, each of which is arranged into a tree of versions. Source components are so-called because they cannot be generated automatically by the system. Users typically create source components using tools such as editors. However, a binary program that is imported into the database from elsewhere is also a source component in this context. All other components, i.e. those that can be generated automatically by the system, are *derived*. Components, once created, are immutable. Tools used to construct derived components are themselves modelled as components and versioned in the same way. Systems are built by executing a functional program in a *system modelling* language. These properties provide the basis for reliable and repeatable system builds.

Unlike Vesta, in which the package structure is hard-wired, a Forest package is just a particular subtype of the `Component` class. The operations available on a package are specified as ordinary C++ methods, and each executes as a transaction. A package version can bind *any* subtype of `Component`, although to date we have not exploited this flexibility. Variations of the package type could be defined (for example to implement different organizations of versions), as indeed could quite different structures, and all could coexist in the same database. This flexibility and extensibility is one of the clear benefits of the OODB approach.

Following Vesta, the component type that we currently bind to a package version is a special kind of container that we call a `BuildableFolder`. This is similar to a file system directory, in that it contains a set of heterogeneous (source) components. However, it also plays a key role in the system building process in that it is also *input* to the system builder. In this role a `BuildableFolder` acts as a module in the system modelling language. The source components act as constant values that are the inputs to the algorithm that computes (builds) the system. Large systems are built up by connecting package versions together using an *import* mechanism. The essential feature of the import mechanism is that it is not intensional; each `BuildableFolder` precisely and permanently identifies the other folders that it imports. The details of the system modelling process are beyond the scope of this paper; the mechanisms are similar to those described for Vesta, but the details are rather different.

Development typically proceeds by the user *checking out* the latest version of a package. This produces a new branch on the version tree and a new `BuildableFolder`, which initially shares the components contained in the folder that was the subject of the checkout operation. The package components, which can be of arbitrary type, are then evolved by appropriate editors. We use the term *evolved* rather than *modified* to emphasize that any editing must take place on copies. We expect editors to exploit the immutability provided by the system and share as much as possible of original components when constructing copies.[1]

Coordination of component editors is overseen by a special editor for a `BuildableFolder`; this editor ensures, amongst other things, that the entire folder is saved as a group, thus versioning the entire configuration. Once a version is saved, which involves committing any new components to the database, it can be built. Therefore, it is important that the group save operation be fast, in user time, and our experience with the database is positive in this regard. Eventually a version of this checkout branch, usually the last, is *checked in* by creating a new version on the main branch. This involves no copying or building, since it merely serves to associate a new version name with the same folder.

4.4 Fine-grain Components

Some components in a `BuildableFolder` have internal structure that is made visible through the interface to the component. Since this structure corresponds to the sub-file granularity in a file-based system, we refer to such components as *fine-grain*. Most current program editors are text-based, so typical source components are not structured, but we expect this to change as object-based systems and compound documents become more widespread. In contrast, it is somewhat easier to represent derived components, those created by tools during a system build, as structured objects by way of wrappers and similar techniques. In Forest, these are defined as appropriate subtypes of the `Component` class, tailored to model the abstract objects they represent.

For example, we specify the input to the parser for a programming language as a `TokenSequence` component that represents a sequence of tokens in that language. This component might be produced by a lexer from a text component or it might be manipulated directly by a language sensitive editor. Either way, the parser, as well as all downstream tools in the compilation pipeline, are only interested in the elements of the token sequence, and are unconcerned that it might be derived from or be part of another component. This pre-

1. This assumes that components are structured so that unmodified substructure is shareable.

cise modelling of the compilation pipeline provides opportunities to avoid certain phases. For example, a change to a comment or other annotation, such as a graphic, in the original source component, would only require re-lexing.

4.5 Uids and Abstract Values

Two particularly important attributes of a component are its *uid* and its *value*. Each component is assigned a globally unique identifier (uid) that denotes it for all time.[1] The uid acts much like a pointer or reference in a programming language. Forest uids are 128 bits wide, part of which encodes the database in which the component is allocated. The uid provides a guaranteed way to identify a component, in contrast to the facilities offered by the UNIX file system. The uid also provides a sure way to determine if two references are to the same object.

However, uid comparison sometimes draws too fine a distinction. Two components with different uids may in fact represent the same abstract value, for example the same sequence of tokens in the `TokenSequence` component of the previous section. The abstract value attribute is represented by a *fingerprint* [13]. A fingerprint is an opaque bit sequence, similar to a hash code, that encodes value probabilistically. Two components with the same value have the same fingerprint. Conversely, if two components have different values, then, with extremely high probability, they have different fingerprints.[2] Each component implementation can provide its own definition of the value attribute; the default, inherited, implementation uses the uid, which is safe but conservative. Fingerprints can be combined efficiently, which is how the abstract value of structured components is computed. Component implementations are free to cache the fingerprint or compute it on demand. The system builder deals with components through their abstract values because, in the system modelling language, object equality is defined as equality of abstract values.

4.6 Copy or Share?

The combination of immutability and abstract value provides many benefits, not least the freedom to copy or share components at will. For example, a system that is built in a database at a developer site can be copied to a database at a client site. Provided that client and developer share the same environment with respect to the dependencies of the copied system, no rebuilding is necessary because, even though the copied components will have different uids, the abstract values of the components will not have changed. If, on the other hand, the client uses a different version of the compiler, then the affected components will be rebuilt transparently; the system takes full responsibility for the construction and location of derived components.

The ability to share components reliably has two major benefits. First, it permits components to be constructed as incremental modifications to existing components. This is similar to the way in which versioning systems such as RCS [14] use "diffs" to encode changes to text files, but it differs by being proactive rather than retroactive. In addition, arbitrary component types can be constructed incrementally in the Forest environment, leaving details to component implementations. The second benefit is the ability to share large derived components such as programs, libraries, or entire subsystems among many versions and many users. This can dramatically reduce overall storage requirements, making the system competitive or better than current file-based environments.

5 Storage Management and Locking

We turn now to the implementation of the configuration management system, and in particular to the impact ObjectStore has on the Forest/C++ programmer. This is felt in two main areas: storage management and data locking. We will describe the mechanisms and policies that we have developed to minimize this impact.

5.1 Allocation Mechanisms

As noted earlier, objects are allocated in an ObjectStore database using overloaded variants of the C++ `operator new`. The C++ language permits optional *placement* arguments to `new`; ObjectStore uses these to control where in the database an object is allocated, using two basic placement abstractions: *segments* and *clusters*.

Segments support large scale structuring of the database and provide hooks for various policies to improve performance. For example, whole segments can be transferred from the server in one request. Segments can be of unlimited size.

A segment can be divided into clusters, which are essentially abstract pages. A cluster has a minimum and a maximum size, the minimum being one page. ObjectStore's locking is actually page-based, so objects in dif-

1. In principle the uid could change, provided the change appeared atomic to all clients, but the difficulty in locating all references makes this impractical.
2. The probability can be increased by adding more bits to the fingerprint representation; we use 64.

ferent clusters cannot cause lock conflicts. Clusters also provide for locality of reference for groups of small objects. Since clusters have a maximum size, a client must always be prepared for an allocation request to fail, requiring the cluster to be extended or a new one to be allocated. Objects larger than the maximum cluster size cannot be allocated in a cluster at all.

It would be unworkable for each allocation site to deal with all these issues, so in Forest we abstract placement data into a C++ class called `DBNewPlace`. Although this successfully encapsulates the ObjectStore placement information, a client must still acquire or generate an instance of this class in order to allocate any objects. Fortunately ObjectStore provides enquiry methods that return the cluster or segment in which an existing object is allocated. Provided such an object is at hand, it is easy to generate a `DBNewPlace` that will cause allocation with the same placement. In practice, almost all objects are associated with some container object, and allocation with the same placement as the container is usually the right thing to do. Since such allocations typically take place inside a method of the container object, the C++ `this` pointer can be used to generate the placement. In cases where a `this` pointer is not available, such as static member functions, a `DBNewPlace` instance must be passed as an explicit argument.

5.2 Allocation Policy

The `DBNewPlace` class provides the mechanism for abstracting the placement information, but who provides the policy? Our experience bears out Object Design's advice: unless this is carefully planned, lock conflicts, poor locality of reference, and consequent inadequate performance are inevitable.

The versioned, mostly immutable nature of a Forest database is of considerable help in establishing appropriate policy. By design, conflict can only occur on the mutable components that represent the leading edge of development. Recall that development by an individual user takes place on a package. Often two or more packages may be involved, for example a package that provides a library of code and an application package which uses the library. Different individuals will typically be working on different packages, ultimately integrating their work by way of packages that denote subsystems.[1]

Each Forest package is allocated in a separate segment, so all source components in all versions of the package are allocated together. Lock conflicts thus cannot arise from modifications to different packages. In addition, when a particular version of a package is built, all new derived components are allocated in a fresh segment. So concurrent builds of different versions of the same package also cannot conflict. Should two programmers be working on different branches (versions) of the same package, conflict can occur while creating new source components. However, this happens in user-time, and the lock is only held for the time it takes to write the new component. If such conflicts proved to be problematic, it would be relatively straightforward to allocate extra segments, for example, for each version branch. Only the code that creates new package versions would be affected by this change; editors and other mutators would simply inherit the appropriate placement.

Owing to the size constraints, we find clusters much less useful than segments; we use clusters only for components whose size we can predict with confidence.

5.3 Uid Generation

The most significant hot spot in the database is the data structure that deals with uids. At a minimum the *next-uid* value must be updated each time a component is created. The data structures that provide a mapping from a uid to the associated component are also potential hot spots. Evidently this data must be clustered separately, otherwise read-only navigation of the database will immediately conflict with mutating.

In fact, as the measurements in section 7 show, the conflicts that arise when multiple processes are mutating the database quickly cause serious performance problems. To address this problem we have made it possible for uid generation to be distributed amongst components in the database. Rather than hard-wire a particular distribution, any component type can opt to be a uid-generator by inheriting and implementing the `UidGenerator` interface. This interface defines a protocol that permits a hierarchy of components to collectively manage the uid space in one database.

Initially the only uid generator is at the root component, which also serves as the entry point to a database.[2] A component can register with the root and acquire a portion of the uid space that it then manages. Any component can be asked for its uid-generator. Since allocation and uid generation go together, we extend the `DBNewPlace` class to carry both the database placement and uid-generator instance. There are no constraints on how a component implements the `UidGenerator` interface; the protocol provides for the enumeration of the components under its control. In principle this hierarchy could be of any depth. In practice, we have only used a

1. Vesta coined the term *umbrella package* for this purpose.

2. It acts like "/" in the UNIX file system.

depth of two, by making each package be a uid generator that registers with the root component.

The contention caused by the uid structures could be avoided by binding our implementation more tightly to ObjectStore, since each persistent C++ object has a unique persistent address.[1] However, an initial goal of the project was to avoid too much dependence on implementation details of any one database, hence our own uid layer. Since the uid representation is hidden from Forest clients, we should be able to perform this optimization in the future. Nevertheless, the uid structure is just a particular example of an *index*, which occur repeatedly in database applications. A consequence of the distributed shared memory model is that such indices are certain hot spots if the update frequency is high.

5.4 Locking

Careful clustering and distributed uid generation does much to alleviate the potential for lock conflicts in Forest, but it does not permit the programmer to ignore the issue completely. In particular, any process that involves user-interaction or indeterminate delay cannot be undertaken inside a transaction without the risk of locking out another process. In Forest, this issue manifests itself most obviously in the user interface to the configuration management system. Much user activity involves browsing the package structure, using read-only transactions, with occasional bursts of editing and building that require update transactions.

In order to avoid holding a read lock that might prevent an update, user-interface code must use short transactions for database navigation. Pointers that represent navigational context must be converted to ObjectStore references, which were alluded to in section 3. ObjectStore provides a variety of reference types, with varying costs and functionality. All are represented in C++ as template types and behave like *smart pointers* to the real object type. For the most part this fiction succeeds, but the C++ type system sometimes falls short. For example, if `B` is a derived class of `A`, then a `B*` is assignable to an `A*`. However, there is no such relation between a `DBRef<B>` and a `DBRef<A>`, where `DBRef<T>` is a reference to a type `T`.

One particular trap for the unwary involves closing a transaction within a method of an object whose `this` pointer becomes invalid when the transaction ends. This is analogous to sawing off the branch on which one is sitting. The solution is to use functions or transient objects to scope transactions.

1. The public interface provides this as a text string that encodes the database, segment and offset.

5.5 How much to Store Persistently?

One reason that database-based environments have a reputation for storage excess is that the data structures associated with compilers and related tools can be very large. A good example is a program represented as an Abstract Syntax Tree (AST). Another is the debugging information in typical UNIX object files.[2]

Forest solves this problem with two distinct mechanisms:

- Separation of interface and implementation; and
- Aggressive sharing of substructure.

The second mechanism, discussed earlier, deserves further emphasis. The intrinsic immutability of Forest components enables common structure to be shared, confident in the knowledge that it cannot change.

The value of physically separating interface and implementation is well understood in the abstract data type programming communities. In the C++ community physical separation by way of abstract base classes is uncommon, perhaps because of efficiency concerns and poor language and compiler support.[3] In a database context, efficiency concerns are clearly secondary to the flexibility that abstract interfaces provide. Forest components are defined and accessed through interfaces defined as C++ abstract base classes. These abstract interfaces show no implementation detail: all methods are pure virtual, and no data members are permitted. The implementor enjoys complete freedom to represent the abstract state of the component in the most convenient way. The actual implementation takes the form of a class that is derived from the abstract class and contains the physical data members.

When coupled with immutability, this separation of interface and implementation becomes a powerful tool to control database size. Consider the AST example mentioned previously. An AST, when annotated with semantic information, is a large data structure that is not obviously worth storing persistently in its entirety. Fortunately, we can easily establish the minimum amount that must be stored: the source component from which the AST was generated, any ASTs that correspond to imported (included) source components, the tool responsible for the AST generation, and any environmental information, such as the target machine,[4] upon which the generation depended. Since, by definition,

2. This is particularly true for C++ code because language semantics makes it difficult to share information from include files.
3. C++ does not treat abstract base classes specially, and some implementations actually penalize their use.
4. Represented, of course, as a component.

these dependent components are immutable, references to these components are all that need be stored persistently. The full AST can be generated in transient memory on demand, and cached in a table keyed by its abstract value (fingerprint).

At the other extreme we might choose to store the entire AST persistently. Abstract base classes, combined with ObjectStore's pointer transparency, leave clients unaware of this transition, other than possible delays when the AST must be regenerated in transient memory. Evidently we have reduced this problem to a classic space-time trade-off. Indeed, it is possible to create both kinds of components in the same database and therefore measure the trade-off for a given processor/network/disk combination, all other factors being held equal. Given the continuing divergence in processor and disk speeds, generating infrequently used data on demand in transient memory seems a sound strategy. Note that in either case the same amount of virtual memory is required for the AST; either it is mapped from the database or allocated transiently by the process.

5.6 Garbage Collection

The high degree of sharing and the concurrent nature of data access makes manual storage management impractical in a Forest database. Fortunately the configuration management framework makes automatic garbage collection a practical possibility. First we must explain the different ways in which garbage can occur in a database. Although most components are immutable, there are exceptions.

- Derived components that result from building a package version can be discarded at will since, by definition, they can be recreated reliably from source components when needed.
- Building typically creates garbage in the form of intermediaries, for example components analogous to compiler intermediate files. These become garbage because they are not reachable from any package.
- Users can delete old package versions, causing some source components, modulo sharing, to become garbage. Since this is a potentially dangerous operation, and since derived components dominate storage costs, it is not clear that source deletion is really necessary.

Garbage collection is currently supported by reference counts on components. Reference counting of derived components is mostly handled automatically by the system builder. However, editors and other tools that manipulate structured components, such as the versioning system, are required to handle reference counting manually. The reference counting interface is part of the `Component` class, but the implementation is private and could be altered without affecting clients.

Since components may reference components in other databases, we have, in general, a distributed garbage collection problem. Each database occupies a distinct piece of the 128-bit wide address space, so there is no conceptual difficulty in garbage collecting a large enough portion of this address space to bound all inter-database references. Note that manipulation of reference counts and garbage collection are the only ways in which a distributed update transaction can occur in Forest. Given that the system builder is based on the abstract value of immutable components rather than uids, and that we can copy any package version from one database to another at will, it is not clear that inter-database references should be encouraged. We expect that a group of collaborating programmers will share a single database, and we postulate that sharing among larger groups might be served best by copying. However, this a matter for further research.

6 Safety

The shared memory model creates the potential for corruption by rogue applications of critical data structures, for example, those that underpin the folder structure and versioning system,. The solutions are either to abandon the shared memory model or to rely on the safety of the programming language in which the system is implemented. Cedar [15] is a classic example of the latter approach.

The potential for corruption is real, since C++ is unsafe, but ObjectStore provides some mechanisms to mitigate this. For example, a program fault is caught and translated into an exception that aborts the transaction, leaving the database unscathed. It is also possible to leverage the type information stored with the database to check the validity of pointers prior to committing a modified page.

In the long term we would prefer a safer implementation language. Compiler support for safe C++ [16], with garbage collection for transient objects, would be a viable alternative.

7 Performance Measurements

Although we have not yet put the prototype system into everyday use, early indications are encouraging. With no tuning at all, the prototype performs quite adequately. We intend to develop the prototype into a working system that we will use ourselves for everyday development.

Meanwhile, in order to estimate the performance of the environment in real use, we have developed a test program that simulates a user performing a set of editing operations. The test database, characteristic of a typical medium-sized project, contains approximately 2500 text components, mostly C++ source files, imported from a UNIX file system. These are distributed among 216 packages, each of which corresponds to a directory in the file system. The initial size of the database is approximately 18Mb.

The program simulates a user checking out a package at random, choosing a random number of components of that package to edit, editing them (by making copies), checking the package in, and repeating this process for a given number of iterations. Unlike real users, the test program performs edits in zero time and so provides a somewhat more stressful test than would occur in real use.

Each instance of the program simulates one user. By running the program on several machines, all accessing a shared database on the server, we can discover how the system scales as users are added. In particular we can measure the effect of making each package a uid generator. All times are wall clock since, in a distributed system, that is the only measurement of relevance to users.

The experiments were run on dual-processor SPARCstation™ 10 client machines with 64Mb of memory, connected by a 10 Megabit Ethernet® to a six-processor SPARCcenter™ 2000 running a beta version of the ObjectStore 4.0 database server. The database was stored in a UNIX file, a convenient feature of ObjectStore, but one that obviously limits performance in comparison to storing the database on a raw disk.[1] Client cache size, the amount of virtual memory available for mapping in database pages, was set to the default value of 8Mb. For this benchmark, which only accesses about 0.5Mb of data, the cache size is not an issue.

The results break down the run time into six activities: initial scan, checkout, checkin, analyzing and constructing a new `BuildableFolder`, reading the text component data, and writing it back in the new component. Each of these operations is implemented as an independent transaction to maximize the potential for concurrency. The initial scan walks the top-level folder structure and counts the number of packages. As a side-effect this partially warms each client cache. Table 1 shows the results when the database contains a single uid-generator. These clearly confirm that contention for write access to data is disastrous in the distributed shared memory model. For operations that require object creation, it is as if each new client adds its load to every other client machine. In other words, no benefit is accrued from distributing the processing among multiple client machines.

Table 2 shows the effect of distributing the uid generation amongst the packages. These results are much more encouraging. In particular the sum of the checkout and write times only doubles as we go from one to eight clients, because of the much lower contention for locks. The consistent times for the scanning and reading phases indicates that the database server is affected only slightly by read-only clients and that the client side caching is effective. Clearly, as the number of clients increases, the probability that the same package will be chosen for editing by more than one client increases, and this does indeed occur during this experiment. This causes more communication between the server and clients in order to reclaim locks, and therefore increases the real-time delay for the affected client.

As noted earlier, the critical measure is the time taken to write back modified components and check in a new package version. In this experiment an average of five components are modified in each checkout/edit/checkin cycle. The time to commit the updates ranges from approximately five to twelve seconds. For the common case, a modification to one file, the results suggest a range of two to five seconds. While we would prefer an upper bound on the order of one second, these figures are acceptable given the prototype nature of the system.

8 Related Work

The Vesta system [12] [17] [18] [19] clearly demonstrated that configuration management could be placed on a firm foundation through the use of immutable components and modular system modelling. The Vesta repository, however, remained essentially file-based, and was implemented on top of an existing file system. Tools (*bridges* in Vesta parlance) were still required to pickle language level objects into repository files and the caches for the builder were also stored in this way. Much of the implementation was concerned with implementing key transactional operations in terms of the underlying file-system. The OODB infrastructure provides this directly, while also supplying more flexibility and extensibility, and a consistent object model from the top-level structure of the repository to the fine-grained structure used by tools. To illustrate the leverage of the OODB, it is stated in [18] that the Vesta repository took less than one person year to implement. In contrast the Forest configuration management subsystem took less than one person month to build.

1. ObjectStore can also store databases on a raw file system.

Table 1: Simulation Results with a Single UID Generator

	Number of Concurrent Users							
Operation	1	2	3	4	5	6	7	8
Initial Scan	36[a]	40	39	44	45	53	57	106
Checkout	21	39	94	114	155	157	195	226
Checkin	16	36	83	92	123	172	179	217
Folder Edit	24	60	92	141	152	153	180	242
Read	29	31	29	30	30	32	30	33
Write	105	162	372	526	682	814	940	1145

a. Measurements are in seconds of wall clock time for 25 iterations.

Table 2: Simulation Results with Multiple UID Generators

	Number of Concurrent Users							
Operation	1	2	3	4	5	6	7	8
Initial Scan	44[a]	43	42	44	48	52	60	65
Checkout	21	26	27	37	49	51	60	76
Checkin	16	16	21	24	22	26	28	31
Folder Edit	21	23	26	30	34	37	37	44
Read	30	30	30	31	32	34	33	37
Write	98	109	127	134	136	181	186	222

a. Measurements are in seconds of wall clock time for 25 iterations.

Vesta was itself influenced by the Cedar System Modeler [20]. The Cosmos [21] project also bases its configuration management on immutable objects.

PCTE [1] includes a database management system, OMS [22], which provides an entity-relationship model with object-oriented extensions, but it was not designed to support fine-grained components. The PACT project [23] developed configuration management services on top of OMS.

ClearCase® [24] and DSEE [25] both support a versioned file system with *views* to select particular versions in individual user contexts. Both support integrated system building. Clearcase is a hybrid system, storing some data in conventional files but storing versioning information and other attributes, some of which can be user-defined, in a database.

Magnusson et. al. [26] describe a revision control system for fine-grained, tree-structured, documents implemented with a client-server model and a specialized database. Their storage model for documents is very similar to our scheme for `BuildableFolders`, except that their documents are strictly tree-structured, whereas our import components produce graph-structured configurations. Because our versioning system is independent of component type, we believe that it can be re-used to represent arbitrary versioned documents. We share their view that a change to an internal node in a document requires the entire document to be revised.

ObjectStore also provides generic version management for arbitrary C++ objects, using a binary differencing mechanism for deltas between versions. References (pointers) to other objects *float* to an appropriate version as pages are mapped in, based on a selection mechanism

that is similar to the view selection of ClearCase. This contrasts with the immutable configuration bindings that characterize the Vesta approach. In addition, checkout/checkin is tied to a hierarchy of workspaces, with the understanding that changes will eventually propagate from the leaves to the root of the hierarchy. This approach, often referred to as the *copy-modify-merge* model, is also taken by Teamware [27], which uses a combination of whole directory trees and SCCS [28] files to represent a configuration. In contrast the Vesta approach tries to minimize the need for copy-modify-merge by using modularization and interfaces between subsystems. We conjecture that both approaches have a role to play in configuration management, but with copy-modify-merge best restricted to fine-grained components for which modularization is not applicable.

Onodera [29] describes experience representing fine-grain C++ program information in a database, also using ObjectStore. The results show that a more global approach to storing information on multiple programs does reduce overall space requirements. It is also noted that pointer-based structures are inherently less space efficient than the tight encodings in current file formats. We believe that our approach of separating interface and implementation and the judicious use of transient objects, can maintain an interface that is convenient for programming yet space efficient in the database.

Baker [30] has argued that functional objects can simplify the construction of distributed and parallel computations, and his conclusions echo many of the ideas of the Vesta approach.

9 Conclusions

We have described the design and implementation of a prototype modern configuration management system. It follows the approach pioneered by Vesta, but is implemented on a commercially available object-oriented database.

Our experience suggests that transacted data, combined with typed, immutable components, provide an excellent basis for a software development environment. Many problems of scale that were experienced by earlier efforts can be solved by the pervasive application of incremental techniques. These are made possible by the combination of immutability and abstract interfaces to objects.

Our performance measurements suggest that it is no longer necessary to build a special purpose repository to support an SDE, and that the current generation of object-oriented databases can provide an adequate infrastructure, provided applications pay appropriate attention to object clustering.

10 Future Plans

Our future plans are mainly in two areas. First, we intend to evolve the prototype into a system that we can use for everyday programming, including the development of the system itself. Our current prototype supports an experimental evolution of C++ [31], which afforded us the luxury of designing the compiler to exploit the database directly. A working system for ANSI C++ will be somewhat more constrained, but we expect at least to replace the use of ad hoc file databases for derived information. For unmodifiable tools, Vesta's bridge techniques [17] are directly applicable.

The second area of research is to expand and exploit the use of fine-grained components in all areas of the software development environment. We plan to investigate the integration of structured document editors with the configuration management system. We are also interested in exploiting component abstract values to achieve build-avoidance at a finer grain, using the basic mechanisms of the system builder.

11 Acknowledgments

Jon Gibbons designed and implemented a prototype version of the Forest component system. Ted Goldstein designed the `DBNewPlace` interface. Roy Levin provided helpful insight and discussions on Vesta.

12 Trademarks

13 References

[1] An Overview of PCTE and PCTE+, Gerard Boudier, Ferdinando Gallo, Regis Minot and Ian Thomas, *Proceedings of the ACM/SIGSOFT Software Engineering Symposium on Practical Software Development Environments,* Boston, Massachusetts, November 1988, 248-257.

[2] Types and Persistence in Database Programming

Languages, Malcom P. Atkinson and O. Peter Buneman, *ACM Computing Surveys* 19,2 (June 1987) 105-190.

[3] The ObjectStore Database System, Charles Lamb. Jack Orenstein and Dan Weinreb, *Communications of the ACM* 4,10 (October 1991) 50-63.

[4] *The Annotated C++ Reference Manual*, Margaret A. Ellis and Bjarne Stroustrop, Addison-Wesley, Reading, Massachusetts, 1990.

[5] Make—A Program for Maintaining Computer Programs, Stuart I. Feldman, *Software--Practice & Experience* 9,3 (March 1979) 255-265.

[6] *The Common Object Request Broker: Architecture and Specification*, Object Management Group. Document No. 91.12.1, 1991.

[7] *Inside OLE 2*, Kraig Brockschmidt, Microsoft Press, ISBN 1-55615-618-9. 1994.

[8] A Simple and Efficient Implementation for Small Databases, Birrel et al., *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, August 1987.

[9] A Study of Pickling Emphasizing C++, Daniel Craft, Olivetti Software Technology Laboratory Technical Report STL-89-2, September 1989.

[10] An Extensible Programming Environment for Modula-3, Mick Jordan. *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, Irvine, California, December 1990. 66-76.

[11] *Smalltalk-80, The Language and its Implementation*, Adele Goldberg and David Robson, Addison-Wesley, Reading, Massachusetts, 1983.

[12] The Vesta Approach to Configuration Management, Roy Levin and Paul McJones, DEC Systems Research Center TR 105, June 1993.

[13] Some applications of Rabin's fingerprinting method, Capocelli et al. (ed), *Sequences II: Methods in Communication, Security and Computer Science*, Springer-Verlag, New York. 1991.

[14] Design, Implementation, and Evaluation of a Revision Control System, Walter F. Tichy. *Proceedings 6th International Conference on Software Engineering*, Tokyo, Japan, September 1982, 58-67.

[15] A Structural View of the Cedar Programming Environment, Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach and Robert B. Hagmann. *ACM Transactions on Programming Languages and Systems* 8,4 (October 1986) 419-490.

[16] Safe, Efficient Garbage Collection for C++, John R. Ellis and David L. Detlefs. *USENIX C++ Conference Proceedings*, Cambridge Massachusetts, April 1994, 143-177.

[17] Bridges: Tools to Extend the Vesta Configuration Management System, Mark R. Brown and John R. Ellis, DEC Systems Research Center TR 108, June 1993.

[18] The Vesta Repository: A File System Extension for Software Development, Sheng-Yang Chin and Roy Levin, DEC Systems Research Center TR 106, June 1993.

[19] The Vesta Language for Configuration Management. Christine B. Hanna and Roy Levin, DEC Systems Research Center TR 107, June 1993.

[20] Practical Use of a Polymorphic Applicative Language, Butler W. Lampson, and Eric E. Schmidt. *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1983, 237-255.

[21] A Unifying Model for Consistent Distributed Software Development Environments, J. Walpole, G. S. Blair, J. Malik and J. R. Nicol, *Proceedings of the ACM/SIGSOFT Software Engineering Symposium on Practical Software Development Environments*, Boston, Massachusetts, November 1988, 183-190.

[22] The Object Management System of PCTE as a Software Engineering Database Management System, Ferdinando Gallo, Regis Minot and Ian Thomas, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, California, December 1986, 12-15.

[23] Configuration Management in the PACT Software Engineering Environment, Ian Simmonds. *Proceedings of the 2nd International Workshop on Software Configuration Management*, Princeton, New Jersey, October 1989, 118-121.

[24] *ClearCase Concepts Manual*, Atria Software, 1992.

[25] Computer-Aided Software Engineering in a Distributed Workstation Environment, David B. Leblang and Robert P. Chase, Jr., *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development*

Environments, Pittsburgh, Pennsylvania, April 1984, 104-112.

[26] Fine-grained Revision Control for Collaborative Software Development. B. Magnusson, U. Asklund, S. Minör, Lund Institute of Technology, Department of Computer Science, LU-CS-TR:93-112.

[27] *CodeManager User's Guide*, Sun Microsystems Inc., Part No. 801-2169-11.

[28] The Source Code Control System, Marc J. Rochkind, *IEEE Transactions on Software Engineering*, SE-1,4 (December 1975) 364-370.

[29] Experience with Representing C++ Program Information in an Object-Oriented Database, T. Onodera, *Proceedings Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, October 1994, 403-413.

[30] Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same, Henry G. Baker, *ACM OOPS Messenger* 4,4 (October 1993) 2-27.

[31] The Clarity Language Definition, Mick Jordan et. al. (ed) Sun Microsystems Laboratories, Technical Report: *in preparation.*

Debugging Storage Management Problems in Garbage-Collected Environments

David L. Detlefs and Bill Kalsow
Digital Equipment Corporation
Systems Research Center
Palo Alto, CA 94301
{detlefs,kalsow}@pa.dec.com

May 12, 1995

Abstract

Garbage collection does not solve all storage management problems; programs allocate too much garbage, requiring excess collection, and may retain too much storage, causing heaps to grow too large. This paper discusses these problems and presents tools implemented in the SRC Modula-3 system that help solve them.

1 Introduction

Many garbage collection enthusiasts, present authors included, have presented garbage collection as a panacea for all storage management problems. Like all marketing hype, this is something of an exaggeration. This paper discusses storage management problems that occur in garbage-collected systems, and describes some tools used in SRC Modula-3 [4] [6] that aid in detecting and isolating such problems.

2 Problems with automatic storage management

A garbage collector solves the two classic problems of explicit storage management:

1. *dangling pointers*, where a block of storage is deallocated too early, while pointers to the block are still in use. If the block is reallocated, different parts of the program will be using the same region of memory for different purposes, with disastrous results.

2. *storage leaks*, where blocks of storage are allocated but never deallocated. If this happens repeatedly in a long-running program, the program's memory requirements grow without bound. This problem is the converse of dangling pointers.

Tools like Purify [7] help identify these problems, but further programming is necessary to solve them. However, when garbage collection is used, these problems never occur.

If garbage collection solves these problems, then what could go wrong? More than enough, as we shall see. Excessive allocation may cause overly frequent collection. Moreover, even with collection, the heap may grow too large. There are a variety of causes for surprisingly large heaps: data structures that were designed without an upper bound on their size, references to "dead" heap objects that are hidden behind abstraction boundaries, and references

hidden by the underlying compilation and runtime system. We consider each these problems in turn.

2.1 Excessive allocation

If a program allocates a great deal of storage for short-term use, it creates a significant amount of garbage. That garbage must be collected; the more quickly garbage is created, the more often it must be collected. Generational techniques can help greatly in decreasing the cost of collecting such short-lived garbage. However, it is still possible to optimize the performance of most garbage-collected programs by locally reusing storage for the most frequently-allocated types, thereby avoiding garbage-collector overhead. Of course, such techniques have the same dangers as explicit storage management.

2.2 Unbounded data structures

A garbage collector collects storage that is not reachable from the root set (the stacks and global variables) of the program. If the amount of reachable storage increases monotonically over time, a long-running program will still run out of memory, even with garbage collection. It is surprisingly common for programmers of long-running systems to create data structures that grow without bound. For example, programs often use caches to avoid redundant computation. If a program was originally used in a "short-lived" context, that is, it was used to compute a result and then exit, every result may have been cached. If this same program is converted for use in a "long-lived" server context, or is used on much larger input problems, then this strategy is unacceptable; a policy and mechanism for regulating the cache size must be added. This may sound obvious, but if the program is large and is developed by many programmers, it may be difficult to pinpoint all such data structures. Some may occur in unfamiliar libraries, perhaps written by third parties, and perhaps available only in object form.

2.3 References hidden by abstraction

A data structure whose concrete state references a heap object while its abstract state does not may also cause the heap to grow too large. For example, consider the simple stack type whose interface and implementation are shown in figure 1.

The `Pop` procedure removes the top element pointer from the abstract stack. But note that the "removed" pointer remains in the concrete state of the stack; the `elems` array is not modified by `Pop`. So if we pushed 100 pointers to large graph structures, then popped them all, and then didn't use the stack again, the graph structures would be retained as long as the stack was; if the stack were a global variable, this would be for the remainder of the program's lifetime.

This kind of problem can be especially bothersome to pin down, since we are accustomed to thinking of our data types in abstract terms whenever possible. It is therefore necessary in a garbage-collected environment to modify such data structures to keep the references in the abstract and concrete states synchronized. In the example above, we would modify the `Pop` procedure to remove the concrete reference, as shown in figure 2.

2.4 References hidden by the system

A similar problem can occur in places beyond the programmer's control. Consider an execution of a program with procedures, A, B, C, and D. A calls B, whose preamble reserves space on the stack for a local variable x, a pointer to a heap object X. B calls C, but saves on the stack the value of x, which had been in a register, creating the situation shown in figure 3 (part a). C returns, and B returns, leaving the pointer in a dead area of the stack, as shown in figure 3 (part b). At this point there is no problem; if X is otherwise unreferenced, a collection could reclaim it. But A now calls D, which also allocates space on the stack to store a value. Before D stores anything into this location, however, a collection occurs – perhaps D requested a heap allocation. The heap pointer stored by B is dead, but is located in the active area of the stack, as shown in figure 3 (part c). The object X, and all objects reachable from it, will be retained by the collection.

We should note that this problem is probably not too important in single-threaded systems, since any pointer on the stack that causes storage to be retained in one collection is likely be overwritten by stack ac-

```
INTERFACE RefStack;
TYPE
  T <: Public;
  Public = OBJECT
    push(r: REFANY);
    pop(): REFANY;
  END;
END RefStack.
```

```
MODULE RefStack;

REVEAL
  T = Public BRANDED OBJECT
    elems: ARRAY [0..99] OF REFANY;
    sp: INTEGER := 0;
   OVERRIDES
    push := Push;
    pop := Pop;
  END;

PROCEDURE Push(self: T; elem: REFANY) =
   BEGIN
     self.elems[self.sp] := elem; INC(self.sp);
   END Push;

PROCEDURE Pop(self: T): REFANY =
   BEGIN
     DEC(self.sp); RETURN self.elems[self.sp];
   END Pop;

BEGIN END RefStack.
```

Figure 1: References hidden by abstraction.

```
PROCEDURE Pop(self: T): REFANY =
  VAR res: REFANY;
  BEGIN
    DEC(self.sp);
    res := self.elems[self.sp];
    self.elems[self.sp] := NIL;
    RETURN res
  END Pop;
```

Figure 2: Corrected version of `Pop`.

tivity before the next collection occurs. However, in multi-threaded environments, the problem may be more serious. Imagine in the example above that procedure D, instead of triggering a garbage collection, waits on a condition variable that is rarely signalled. The thread executing D will be blocked for some time, perhaps for many garbage collections. Those collections will retain the object X.

Note that this is not a problem confined to *conservative collectors* such as the collectors of Bartlett [1] or Boehm and Weiser [2], which assume any bit pattern in the stack that looks like a pointer is a pointer. Lisp systems using hardware tags are just as vulnerable; the pointer values in the stack locations are perfectly valid pointers — they just aren't live at the time of collection. The Boehm-Weiser collector attempts to prevent this problem; it zeros the part of the stack above the stack pointer on each collection. A complete solution to this problem requires a great deal of cooperation between a garbage collector and

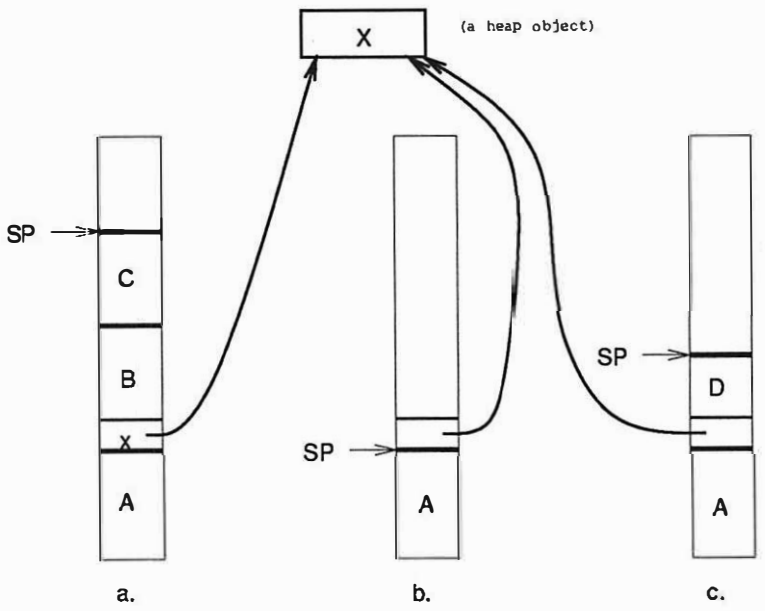


Figure 3: References hidden by the runtime-system.

a compiler. The compiler must either enable the collector to precisely determine which stack values and registers are live at the start of a collection, or generate code to "NIL out" pointer-containing stack locations on procedure entry or exit.

Finally, we should note that while this problem may seem obscure, it does occur in practice. Retention of excess storage was traced to precisely this situation in the first version of the Vesta configuration management system [5]. We know of no certain fix for this problem other than requiring compilers to produce code in which procedures zero-fill their stack frames on entry (or, alternatively, on exit). The obvious performance penalties of these solutions make them somewhat unattractive.

3 Tools

This section describes four tools we have developed to aid programmers both find and fix storage management problems in long-lived garbage-collected programs. These tools are all implemented as part of the runtime system for SRC Modula-3. We give "real-life" examples of the use of each tool. These examples arise from problems encountered in the *Extended Static Checking* (henceforth *ESC*) program verification system being developed at SRC.

3.1 Diagnosing excessive allocation

Excessive allocation causes frequent and potentially intrusive garbage collection. *Shownew* is a tool that allows a user to observe the allocation behavior of a program. Shownew is integrated into the runtime system, so that any SRC Modula-3 program can be passed a special command-line argument that will cause it to run under the control of a Shownew process. Shownew presents a bar graph indicating how much storage of each type is being allocated. A menu allows the user to indicate whether the graph should display the number of objects or bytes allocated, and whether the numbers should indicate totals since the beginning of the program, or only new allocations since the display was last updated.

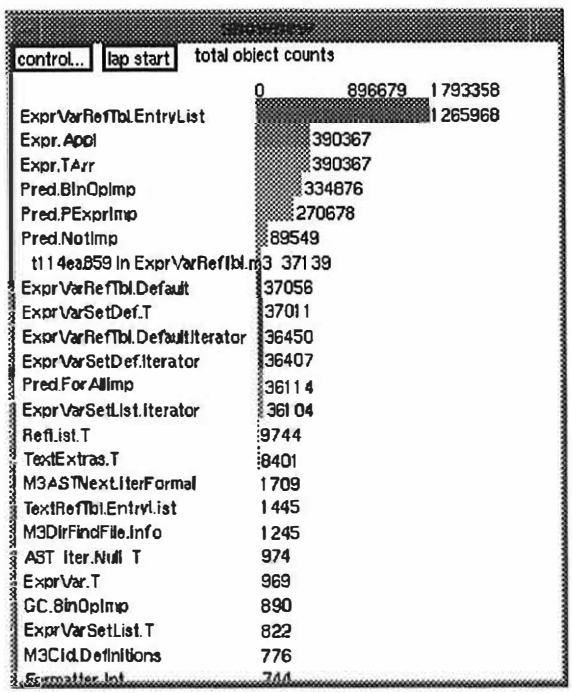


Figure 4: Shownew output (before).

Shownew allows the programmer to pinpoint what types are being allocated most often and might be causing excessive collection.

With Shownew, it is quite easy and almost always worthwhile to determine what types are allocated most frequently in your program. The answers are sometimes surprising, and occasionally represent bugs that are simple to correct. When the ESC system was found to be embarrassingly slow on a new example, we tried Shownew to see if excessive allocation was a problem. The result is shown in Figure 4. (Of course, the bars of the graph appear in color on a color monitor.) The identity of the type allocated most frequently was a complete surprise. `ExprVarRefTbl.EntryList` is an internal type used in the implementation of a set type provided by the Modula-3 library. These allocations were eventually traced to code that applied a variable substitution to a universally quantified formula (a `ForAll`), as shown in figure 5. The argument `exc` is a set of "excluded" variables, variables that are *not* to be substituted. Since a quantifier binds the quantified variables, `ForAllSubst` adds its own quantified variables, `self.vs`, to `exc` before applying the substitution to its body.

It happened that the type `ExprVarSet.T` was implemented using hash tables. The set type's `union` method is non-destructive, so `exc.union(self.vs)` makes a copy of `exc`, adds the elements of `self.vs`, and returns the copy. Note that the copy is discarded as soon as the call to `MkForAll` returns.

We modified this procedure to avoid the copy, as shown in figure 6. The usual way to avoid this copy, and the one we used, is to substitute destructive operations (in which `exc` is modified) for functional operations (in which `exc` is not modified). The call `exc.unionD(self.vs)`, for example, adds the elements of `self.vs` to the set `exc`, modifying its value. Similarly, `diffD` destructively deletes elements from a set.

In this example, several assumptions are necessary to argue the correctness of the transformation. First, `exc` is not being accessed by any concurrently executing threads. Second, it is a precondition of `ForAllSubst` that `exc` and `self.vs` are disjoint. Third, the call `self.body.subst(s, exc)` does not retain a reference to `exc`. Using destructive operations often yields significant performance benefits, but the subtlety of the assumptions necessary to show them correct argues that they should be used sparingly. A tool like Shownew helps identify the most promising targets.

The first picture in figure 7 shows the result of Shownew on this program after the change described above. Note that the `EntryList` type is now only the sixth of the most frequently allocated types; the number of `EntryLists` allocated decreased by more than 95%. Similar transformations resulted in the situation shown in the second picture of that figure, where the absolute numbers of the most frequently allocated types have dropped dramatically.

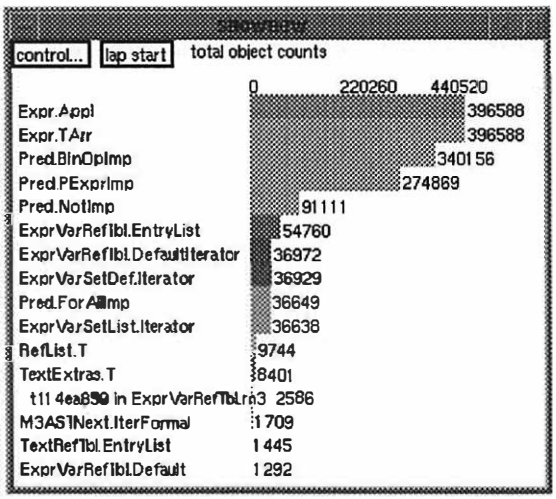
The implementation of Shownew is fairly simple. Each (garbage-collected) heap-allocated object in Modula-3 has a header that contains a *typecode*, a unique integer corresponding to the dynamic type of the object. From a typecode it is easy to find the name, size, and various other attributes of the corresponding type. A `NEW(T)` expression in the source is compiled to a call to the runtime's allocation routine with the typecode of `T` as an argument. When

```
PROCEDURE ForAllSubst(self: ForAll; s: Subst.T;
                      exc: ExprVarSet.T): ForAll =
  BEGIN
    RETURN MkForAll(self.vs, self.body.subst(s, exc.union(self.vs)));
  END ForAllSubst;
```

Figure 5: Excessive allocation (before).

```
PROCEDURE ForAllSubst(self: ForAll; s: Subst.T;
                      exc: ExprVarSet.T): ForAll =
  VAR res: ForAll;
  BEGIN
    exc := exc.unionD(self.vs);
    res := MkForAll(self.vs, self.body.subst(s, exc));
    exc := exc.diffD(self.vs);
    RETURN res;
  END ForAllSubst;
```

Figure 6: Excessive allocation (after).



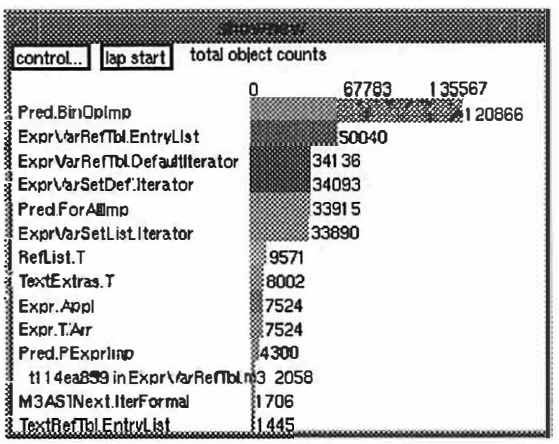


Figure 7: Shownew output (after)

Shownew is being used, the allocator counts the objects allocated of each type, periodically forwarding that infonnation to the Shownew process.

3.2 Excess retained storage

The previous section dealt with allocation. Allocation and heap size are obviously related, since something must be allocated to become part of the heap, but they are also quite different. A type that is allocated often may contribute nothing to the heap size, if all instances quickly become garbage and are collected, while a type that is allocated infrequently, but whose instances are retained, will cause the heap to grow without bound.

The rest of this section introduces three tools used to diagnose problems in the theorem-prover used in ESC. In long-running proofs, the theorem-prover's heap seemed to be growing continually larger, while we expected the storage requirements of the systems to quickly reach a steady state.

3.2.1 RTutils.Heap

The first tool we used was the procedure `RTutils.Heap`, which reports the composition of the heap by data type. If we explicitly run a collection before generating such a report, we obtain

a breakdown by type of live objects in the heap. `RTutils.Heap` has several options that allow the report to be ordered by number of allocated objects or bytes occupied, and to limit the report to the top n types by the requested ranking method. We modified ESC to periodically perform a garbage collection and call `RTutils.Heap`, reporting the top 10 types ranked by bytes occupied. Figure 8 shows these reports near the beginning (but after we would have expected the heap size to stabilize) and end of an execution of the theorem-prover on a relatively small test case.

The heap has grown by almost three megabytes. The bulk of that increase can be attributed to the type `Enode.Parent`, which grew by more than two megabytes.

The addition of calls to `RTutils.Heap` to the source code has helped us identify the first-order term of our problem: we are retaining more `Enode.Parent` objects than we expected. These reports are useful enough that we left this debugging code in permanently, with printing of the reports controlled by an environment variable.

The implementation of `RTutils.Heap` requires the ability to enumerate all the objects in the heap and classify them by type. Recall that Modula-3 heap objects are prefixed by headers containing a typecode that uniquely designates a type. It is therefore simple to examine all heap objects, constructing a table mapping typecodes to object counts and bytes occupied. Finally, the table is sorted by the appropriate metric and printed.

Two additional features of `RTutils.Heap` merit explanation.

First, sometimes a report by typecode is too coarse-grained. For some problems, it is convenient to program in a Lisp-like style within Modula-3. Instances of the type `RefList.T` are singly-linked lists of generic pointers. This type can be used much like a Lisp cons cell. In programs that use such a style, lists are used in a number of different ways. A report saying that the heap contains many `RefList.Ts` may not identify which of the many uses of the type is responsible for the retained storage. In such a situation, the programmer can use the `RTAllocStats` interface. This interface exports a procedure `EnableTrace`, which takes a typecode argument. Once `EnableTrace` is called for a type, the header of each newly allocated object with that type is annotated with a small integer representing the *call site* of the allocation. Abstractly, a call site can be thought of as a snapshot of the stack at the time of the allocation. Practically, it is the sequence of the top n program counter values on the stack, where n defaults to 3 but can be set by the user. The single program counter of the actual allocation is insufficient; in the `RefList` case mentioned above, few `RefList.Ts` are allocated directly by clients of the interface; most are allocated using convenience procedures of the interface like `RefList.Cons` or `RefList.List3`. It would be of little use to discover that most lists were allocated within calls to `Cons`. Figure 9 shows an excerpt from one of these finer-grained reports.

Call-site tracking requires more from the runtime system. The allocation routine must first determine whether tracking is enabled for the type being allocated. If so, it must determine the call site, which requires the ability to interpret a thread stack's contents at runtime; note that this code is highly platform-specific. Because we use "unused" bits in the standard object header to record call sites information, we are currently limited to 256 sites per type. If this limit is reached, allocations at other sites are credited to a site code reserved for other sites, as shown in figure 9.

The second additional feature of `RTutils` solves the opposite problem: when reports by typecode are too fine-grained. For example, ESC processes Modula-3 using M3TK [3], which translates the source code into an abstract syntax tree (AST) that ESC then analyzes. These AST's are made up of many different types; it would be tedious to determine how much space is occupied by such trees from the kind of report described above. However, the types of the tree nodes are organized in a type hierarchy rooted at a generic `AST.NODE` type. `RTutils.Heap` produces a second report for Modula-3 object types that reflects the inheritance hierarchy. Since Modula-3 supports only single inheritance, this hierarchy is a simple tree. In figure 10 it becomes clear that AST nodes occupy about 1.2 megabytes, with different subtypes accounting for different fractions of that total.

The restriction to single inheritance makes the implementation of this report simple. Typecodes are

Near beginning:

```
Code   Count    TotalSize  AvgSize  Name
---- --------- --------- --------- --------------
 270         1   2457616   2457616 Enode.UndoStack
 180      3352    616768       184 Enode.Parent
 173     14623    350952        24 RefList.T
 230         1    160024    160024 Simplex.RatArr2
 294      3804    121728        32 SigTab.EntryList
 233         4     81984     20496 <anon type>
 143      1857     59424        32 Context.Literal
 178       411     59184       144 Enode.Leaf
 161       473     49192       104 MatchingRule.Rule
 196        19     48216      2537 RTHooks.CharBuffer
     --------- ---------
         33714   4505664
```

Near end:

```
Code   Count    TotalSize  AvgSize  Name
---- --------- --------- --------- --------------
 180     16395   3016680       184 Enode.Parent
 270         1   2457616   2457616 Enode.UndoStack
 294     14542    465344        32 SigTab.EntryList
 173     14611    350664        24 RefList.T
 233         4    311360     77840 <anon type>
 230         1    160024    160024 Simplex.RatArr2
 178       852    122688       144 Enode.Leaf
 232         1     64016     64016 Context.TritArray
 143      1790     57280        32 Context.Literal
 161       503     52312       104 MatchingRule.Rule
     --------- ---------
         59951   7684096
```

Figure 8: Use of `RTutils.Heap`

```
Code   Count    TotalSize  AvgSize  Name
---- --------- --------- --------- -------------------------
272          1   2457616   2457616 Enode.UndoStack
173      15416    369984        24 RefList.T
          3436     82464        24    Cons + 16_3c in RefList.m3
                                      SubWork + 16_59c in PredSx.m3
                                      SubWork + 16_4dc in PredSx.m3
          2333     55992        24    Cons + 16_3c in RefList.m3
                                      CopySx + 16_130 in Clause.m3
                                      CopySx + 16_c4 in Clause.m3
...
           465     11160        24    OTHER SITES
...
```

Figure 9: `RTutils.Heap` output broken down by call site.

```
Code   Count    TotalSize  AvgSize  Name
---- --------- --------- --------- -------------------------
...
311      24866   1223512        49     AST.NODE
...
510        964     49208        51          M3AST_AS.STM
...
516        254     12192        48              M3AST_AS_F.Assign_st
...
527         39      2808        72              M3AST_AS_F.For_st
...
```

Figure 10: `RTutils.Heap` output broken down by the type hierarchy.

assigned to types in a pre-order traversal of the type forest, so that the typecodes of the subtypes of a type form a consecutive sequence of integers. The subtype test is therefore a simple range test, making it easy to aggregate the numbers for a type and all its subtypes.

In summary, `RTutils.Heap` answers some of the most basic questions that must be answered when debugging a storage management problem: how big is the heap, what kind of objects are in it, and how many objects of each type are in it? It can answer these questions at finer or coarser grains where necessary.

3.2.2 RTHeapStats.ReportReachable

`RTutils.Heap` identified *what* was filling up the heap, but did not tell us *why* the objects filling the heap were escaping collection. The next tool in our kit helps answer that question. An object is retained by garbage collection only if it is reachable from a *root* of the program, that is, from the stacks, registers, or global variables. The procedure `RTHeapStats.ReportReachable` tells us how many bytes of storage are reachable from the roots, breaking down the roots in various ways. Each global variable in Modula-3 is associated with some module. One list ranks the modules by bytes

reachable from their global variables. A more detailed breakdown ranks the individual global variables by the amount of storage they reach. A second section details storage reachable from thread stacks, again, reporting both at coarse and fine grains. In the coarse grained report, entire stacks are lumped together. In the fine grained report, stack frames are printed along with the storage reachable from individual references contained in those frames.

We modified ESC to call `ReportReachable` periodically. Figure 11 shows excerpts from two reports, one near the beginning of the program, but after we expected the heap size to reach a steady state, and one from near the end.

Comparing the reports identifies the `Enode` module as the major offender. We see also that an object of type `SigTab.Default` in the `Enode` module went from reaching just over one megabyte to almost three and one half. The only global variable of this type in the module was named `sigTab`. Once we had identified `sigTab` as a possible problem, a moment's thought was sufficient to reveal our mistake. The purpose of `sigTab` is to ensure that we never create distinct `Enode.Parents` with identical children. For small examples it was acceptable to remember all the parents ever created. For large examples, however, such unbounded growth leads to overly large heaps. This table is a cache of parent nodes; we needed to invent a cache-management policy for it. We can delete a `Parent` from the table when it is no longer in use. Because of its backtracking search structure, the prover "unwinds" every action it takes; so we can delete a `Parent` from the table when we undo the action that added it in the first place.

We now briefly describe the implementation of `ReportReachable` before continuing with our detective story. Essentially, each line item in the report is the result of a mini-garbage-collection. We allocate a new bit vector large enough to serve as mark bits for all the objects in the heap. To determine the set of objects reachable from some subset r of the roots, we clear the mark bits, then (recursively) mark each object reachable from r, using the mark bits to terminate the recursion. When the mark phase is complete, we count the objects and bytes corresponding to the mark bits. This algorithm requires the ability to enumerate the pointer-containing fields of an arbitrary heap object. Note also that presenting the result of thread stack traversals requires the same run-time interpretation of thread stacks needed for call-site tracking in section 3.2.1.

Note that this process is potentially quite expensive. If a heap contains a large linked data structure that is reachable from many roots, that structure will be traversed and counted once for each of the different root references. In practice, however, the performance of `ReportReachable` seems quite acceptable; the reports shown above did not slow down the the execution of the program greatly.

3.2.3 RTHeapDebug

In the ESC problem we have been discussing, it was fairly easy to identify a problem once `RTHeapStats.ReportReachable` led us to an offending global variable. Sometimes, however, this information may not be sufficient; it may be difficult to determine what paths exist from the offending root to objects of the problematic type. The `RTHeapDebug` interface helps find such paths.

Even in a garbage-collected system, it is sometimes possible to identify a point in the program where one expects an object to become garbage. In our ongoing example, we added code to remove an `Enode.Parent` from `sigTab` when the prover's search backtracked past the point where the parent was first created and added to the table; we expected the parent to be unreachable after it was deleted from the table. `RTHeapDebug` allows the programmer to check such expectations.

Calling `RTHeapDebug.Free(r)` asserts that the reference `r` is unreachable. A call to `RTHeapDebug.CheckHeap` does the equivalent of a garbage-collection, traversing all reachable objects in search of any that have been asserted unreachable. If such an object is found, `CheckHeap` prints a path from a root to the putatively unreachable object.

In the ESC example, we called `RTHeapDebug.Free` to assert that `Enode.Parents` removed from `sigTab` by the undo code were unreachable. We called `CheckHeap` at a later point. We were surprised to find that our assumption was wrong. Figure 12 shows an excerpt of a report from

Near beginning:

```
HEAP: 0x14009a000 .. 0x140c98000 => 11.9 Mbytes

Module globals:
 # objects   # bytes  unit
 ---------  --------  -----------------
     24471   4151816  Enode.m3
     11116    698648  Context.m3
      8390    579608  Clause.m3
       ...

Global variable roots:
 # objects   # bytes         ref type                location
 ---------  --------  ---------- -----------------   -----------------
      8450   3036328  0x140694008 Enode.UndoStack     Enode.m3 + 2224
      9065   1098920  0x1402a2558 SigTab.Default      Enode.m3 + 1352
     12925    703144  0x1400a7990 IntRefTbl.Default   Enode.m3 + 4328
       ...
```

Near end:

```
HEAP: 0x14009a000 .. 0x141244000 => 17.6 Mbytes

Module globals:
 # objects   # bytes  unit
 ---------  --------  -----------------
     43159   6534144  Enode.m3
     11194    757936  Context.m3
      5215    748008  Simplex.m3
       ...

Global variable roots:
 # objects   # bytes         ref type                location
 ---------  --------  ---------- -----------------   -----------------
     27149   3456264  0x1402a2558 SigTab.Default      Enode.m3 + 1352
      8632   3100256  0x140694008 Enode.UndoStack     Enode.m3 + 2224
     13471    727832  0x1400a7990 IntRefTbl.Default   Enode.m3 + 4328
       ...
```

Figure 11: Use of `RTHeapStats.ReportReachable`.

```
Path to 'free' object:
   Ref in root at address 0x14000a770...
   Object of type Enode.UndoStack at address 0x14013c008...
   Free object of type Enode.Parent at address 0x14013eac8...
Path to 'free' object:
   Ref in root at address 0x14000a770...
   Object of type Enode.UndoStack at address 0x14013c008...
   Free object of type Enode.Parent at address 0x14013eae0...
Path to 'free' object:
   Ref in root at address 0x14000a770...
   Object of type Enode.UndoStack at address 0x14013c008...
   Free object of type Enode.Parent at address 0x14013eaf8...
...
```

Figure 12: Use of `RTHeapDebug.CheckHeap`.

`CheckHeap`. There was only one variable in the program of type `Enode.UndoStack`; it is the stack on which undo records are written to implement the backtracking search. Apparently this stack was pointing directly to objects we thought should be unreachable.

A little investigation revealed the problem to be a case of references hidden by abstraction, almost exactly the situation described in the example of section 2.3. We considered the undo stack as only the portion below the stack pointer, but the garbage collector considered the entire array data structure. Undo records above the stack pointer contained pointers to otherwise unreachable objects. To solve the problem we modified the *pop* operations of the various undo stacks in the system to set pointers in dead stack entries to `NIL`. This investigation has purged ESC of its most egregious storage leaks.

It might be argued that the technique embodied by `RTHeapDebug` is regressive, in that it requires the programmer to do the equivalent of explicit storage management. There is some truth to this argument. However, `RTHeapDebug.Free` is needed only for types identified as storage problems, not for all deallocated objects. Calling `RTHeapDebug.Free` for an object that is still accessible causes a graceful error message, while actually deallocating a still-accessible object is likely to cause a confusing dangling pointer bug.

The implementation of `RTHeapDebug` uses the `WeakRef` interface provided by the Modula-3 runtime. A `WeakRef.T` is a reference to an object that does not prevent the object from being collected. `WeakRef` provides a routine for obtaining the "real" reference corresponding to a `WeakRef.T`; this routine returns `NIL` if the referenced object has been collected. `RTHeapDebug` maintains a set of weak references to freed objects, initially empty. `RTHeapDebug.Free(r)` adds a weak reference corresponding to `r` to this set. `RTHeapDebug.CheckHeap` first throws away any elements in the set whose referents have been collected, then searches for paths to the remaining freed objects.

4 Conclusion

We present four classes of storage-management problems that occur even in completely garbage-collected systems: excessive allocation, data structures that grow without bound, references hidden by abstraction boundaries, and references hidden by the system. We have presented tools that aid the diagnosis of such problems. These tools are part of the SRC Modula-3 system, but could easily be adapted for other languages; in fact, some of the tools are not specific to garbage-collected systems.

We feel that the trend towards long-lived server

programs will speed the adoption of garbage-collected systems; the likelihood of pointer-smashes or storage leaks in explicitly managed systems is far too high for applications that must run reliably for long periods. This paper addresses the (more manageable!) storage management problems that remain once such systems have adopted garbage collection.

References

[1] Joel F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, February 1988.

[2] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[3] Mick Jordan. An Extensible Programming Environment for Modula-3. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*. ACM, ACM Press, 1990.

[4] Bill Kalsow. SRC Modula-3 home page. URL http://www.research.digital.com/SRC/modula-3/html/home.html.

[5] Roy Levin and Paul Mcjones. The Vesta Approach to Precise Configuration in Large Software Systems. Research Report 105, Digital Equipment Corporation Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, June 1993.

[6] Greg Nelson, editor. *Systems Programming in Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[7] Pure Software, Los Altos, California. *Purify Version 1.1 Beta A*, 1992. User manual.

Phantom : An Interpreted Language for Distributed Programming

Antony Courtney*
Department of Computer Science
Trinity College Dublin
Ireland

Abstract

The emerging trend in writing distributed applications is to use an object-based RPC system with a statically compiled, object-oriented language. While such a programming environment is adequate for many tasks, object-based RPC systems and statically compiled languages also have certain intrinsic limitations. These limitations become significant when writing applications which are both distributed and interactive (e.g. network information browsers, distributed conferencing systems and collaborative work tools). This paper discusses these limitations, and presents the design of *Phantom*, a new intepreted language for distributed programming. Phantom provides many features found in object-based RPC systems and statically compiled languages, including *automatic marshalling*, *transparent remote procedure call*, *secure authentication* and *concurrency support*. In addition to these traditional features, Phantom's interpreted nature permits the use of certain programming techniques, such as *true object migration*, *remote evaluation*, and *dynamic extensibility*, which are of increasing importance for distributed programming, but which are not available in statically compiled languages and RPC systems. The integration of these features in a single, coherent programming language makes whole new classes of distributed, interactive applications possible.

1 Motivation

1.1 Object-based RPC and Static Compilation

The current trend in programming distributed systems is to use a statically compiled, object-oriented language (such as C++ [Str92]), augmented with an object-based RPC system and associated runtime library (such as CORBA [Gro92] or ILU [JSS94]).

With an object-based RPC system, the programmer specifies the interface to network-accesible objects using an *interface definition language* (IDL). A *protocol compiler* compiles the IDL specification to generate client and server *stub routines* which are linked with the programmer's application code. For each remote object which the application has access to, the

*email: Antony.Courtney@cs.tcd.ie

client stub routines provide a *local surrogate object*. The local surrogate object appears (to the application programmer) like a normal programming language object. However, when the application invokes a method on the local surrogate, the surrogate sends an RPC to the remote server object which actually performs the request, and the local surrogate returns the result of the RPC as if the operation were performed locally. This provides a degree of *location transparency* for the programmer: there is no way of distinguishing method invocation on true objects from method invocation on surrogate objects.

Using a statically compiled, object-oriented language with an object-based RPC system provides a number of benefits over unstructured, transport-layer message passing. In particular, such systems solve heterogeneity problems, provide location transparency, and add structure to the network communication between different programs.

However, object-based RPC systems and statically compiled languages also have some significant deficiencies. For purposes of this paper, the most significant deficiency is that object-based RPC systems do not provide any mechanism for sending code across sites. This excludes certain distributed programming techniques (such as Remote Evaluation [SG90]) and, as the following example will illustrate, also places severe limitations on both the dynamic extensibility and performance of interactive applications.

1.2 An Example from the World-Wide Web

The limitations of object-based RPC systems and statically compiled languages become significant when writing applications which support both remote network access and interactive user interfaces. An illustrative example of these limitations comes from the World-Wide Web (WWW) [BLCGP92].

A typical example of an interactive application for WWW is Lawrence Berkeley Laboratory's "Interactive Frog Dissection Kit" [RJN94], an educational program which uses three dimensional volume rendering to allow a user to graphically explore the anatomical structures of a virtual frog. A screen-shot of the program is shown in figure 1.

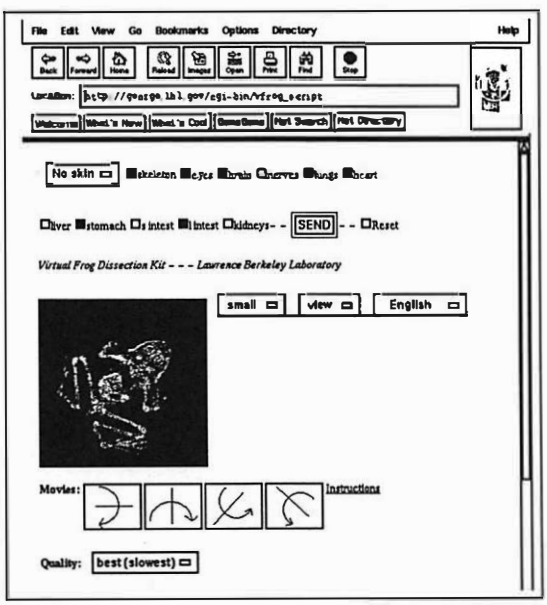


Figure 1: An interactive WWW application.

This application has two parts. There is a *client* part, which displays the visual image of the current view of the frog, and allows the user to change this view by setting various parameters. In the current implementation, the client part is an HTML [BLC93] form, which is displayed and managed by the user's WWW browser program. There is also a *server* part, which runs on a remote, high-performance computer and renders the three-dimensional image of the current view of the frog. In the current implementation, the server is a C program run via WWW's Common Gateway Interface (CGI).

An alternative approach to using HTML forms and CGI for this application would be to use object-based RPC between the interactive client program and the remote rendering server. There are two possibile approaches to doing this:

1. Develop special-purpose client software to access the server, specifically for this application. This gives the programmer the freedom to decide what to compute on the client side, what to compute on the server side, and how much state to keep between the client and the server. The programmer can apply specialised knowledge of the application to optimise interactive response in the client, and minimise network usage between the client and server. However, this approach requires that users obtain and compile the special-purpose client software. This added burden on potential users precludes exactly the sort of casual exploration which has made WWW (and its many interactive applications) so popular.

2. Develop a completely general client program, which can be used for many different interactive applications and which allows remote server programs to create and manage arbitrary GUI objects in a client window. The problem with this approach is that if the client program is sufficiently general, giving the server arbitrary control over the client window, then all responses to interactive events must be handled by the server application. Experience with the X Window System (and its various object-based toolkits) has shown that remote handling of interactive events simply will not yield acceptable response time over networks with higher latencies than a high-speed LAN.

The use of CGI and HTML for such applications has two important benefits over using object-based RPC between the client and the server. First, any WWW browser which supports forms can serve as a client program for any specialised application server. Second, the browser program handles all direct user interaction with the HTML form for the server. This ensures that the client user interface will have adequate interactive response for those GUI events (such as individual key presses) which do not require server interaction.

However, using CGI and HTML to build interactive applications creates its own set of problems:

- The set of interaction mechanisms provided by HTML forms, and their presentation by the WWW browser, is relatively fixed. For example, HTML does not currently provide any mechanism for creating dialog boxes or handling modal interactions.

- The benefit of local interactive response is limited to those applications which fit cleanly into a "forms" model of interaction, such as database clients. Applications which don't fit cleanly into the forms model (i.e. because they require application-specific behaviour for each interactive event) are either impossible to develop or suffer from exceedingly poor interactive response. For example, it would be impossible to develop a soft real-time game using HTML and CGI.

- Items in the GUI presented by a CGI program can not be changed individually; the CGI

program must send a new form back to the client for every minor change in the interface's appearance.

- There is no server-side state maintained; the client must re-transmit the entire form contents to effect any change in the server.

The problem, effectively, is that a general-purpose document specification language (HTML) is being used as a batch-oriented programming language for specifying and controlling Graphical User Interfaces.

We believe that there is an alternative approach to implementing interactive, distributed applications (such as the frog dissection kit), which does not sacrifice generality or interactive response time. Instead of a remote application server transmitting an HTML document to the local browser, our approach is to have the remote application transmit a procedure (written in a general purpose programming language) across the network, which the browser will execute. This procedure will be given complete control over some region of the user's display (e.g. a window), so that the programmer of the application server is not tied to any fixed set of user interface mechanisms. And because the procedure received from the application server is executed locally, interactive response time will not suffer.

1.3 The Phantom Approach

To address the problems inherent in writing applications which support both remote network access and interactive user interfaces, we have developed *Phantom*, a new language and runtime environment for writing distributed applications[1]. The goal of Phantom is to provide a single, powerful infrastructure for developing large-scale, interactive, distributed applications. To meet this goal, Phantom attempts to redress some of the deficiencies of statically compiled languages and RPC systems. Phantom is able to address these deficiencies for the following reasons:

1. *Program code may be transmitted across sites.* This is, perhaps, Phantom's single biggest advantage over statically compiled languages and RPC systems. Phantom provides generalised concepts of *higher-order functions* and *lexical scoping* in the context of a distributed system. The use of higher-order functions provides a semantically sound, easily-understood mechanism for sending small parts of programs (individual procedures and functions) to remote sites for execution. Lexical scoping is used to guarantee that procedures received from remote sites will not have access to any resources or information on the local site which could not be accessed via RPC from the remote site.

2. *The language was designed with distribution in mind.* The Phantom language appears to application programmers much like a conventional programming language. However, certain details of the type system (such as the elimination of arbitrary pointer types) make it possible for the runtime to provide transparent distribution facilities for Phantom code and data.

3. *The language is interpreted.* Using an interpreter for the language makes the run-time environment very flexible. In particular, implementing Phantom as an interpreter makes it easy to support very high-level data types, perform automatic storage management, provide general-purpose higher-order functions, dynamically load and execute Phantom programs, and provide transparent access to code and data at remote sites.

[1] The name *Phantom* derives from the "transparent" nature of the language's distributed programming features.

4. *Distribution is flexible and transparent.* In statically compiled languages and RPC systems, extra programming work is required to specify which values or procedures may be made accessible across the network, and the programmer is burdened with even more work if the data types to be shared are linked or cyclic. In Phantom, all program values are potentially and transparently network accessible. However, the details of distribution are kept under flexible program control, so that programmers may optimise their applications for performance.

While none of the above features is compelling in itself, their integration in a single, simple language provides a powerful environment for constructing distributed applications.

2 Related Work

Distributed programming languages are not a new idea. An excellent overview of other distributed programming languages is given in [BST89], which includes a bibliography of over 200 papers on nearly 100 different languages. Despite the number and variety of these other distributed programming languages, most other distributed languages are intended for harnessing the power of a network of processors as a single parallel processing engine. Furthermore, most other distributed programming languages do not provide any mechanism for transmission of code across sites. The only language we are aware of which provides facilities for transparent distribution of both program code and data is Obliq [Car95].

The distribution model of Phantom uses the same basic model as Obliq. However, there are a number of aspects of Phantom which differentiate it from Obliq:

- Phantom is strongly typed (using structural type equivalence). The adherence to strong typing provides a level of static error checking which is not provided in most other interpreted languages (including Obliq, Tcl [Ous94] and Python [Ros92]). As examples will illustrate, Phantom provides *type-safe implicit declarations*, which limit the overhead of strong typing in an interpreted language.
- Phantom uses a class-based object model, rather than prototypes. This, in conjunction with strong typing, encourages the seperation of interface from implementation of object types.
- Syntactically, Phantom more closely resembles Modula-3.
- Phantom provides a simple access control and authentication mechanism at the language level.
- The Phantom interpreter is implemented purely in ANSI C, and provides a library interface to the Tk toolkit [Ous94].

Although we chose to design a new language, producing a novel new programming language is not the goal of the project. We tried hard to borrow proven mechanisms from other languages and systems rather than invent our own. The language core is based on the syntax and semantics of Modula-3, so that it will be accessible to application programmers with exposure to any successor of Pascal. For distribution, Phantom uses the distributed lexical scoping semantics of Obliq. For access control, Phantom uses a model which is similar to permissions in the UNIX file system [RT74].

3 The Phantom Language: Concepts and Techniques

The Phantom language supports a number of modern imperative programming features, including: *interfaces*, *objects*, *threads*, *garbage collection* and *exceptions*. These are all features of the programming language Modula-3 [Nel91]. Where possible, Phantom borrows Modula-3's syntax and semantics. Phantom extends Modula-3's object model (by adding the concepts of *ownership* and *access control*), and omits many of Modula-3's more complex features (such as *reference types* and *generics*) which would complicate Phantom's distributed semantics. Phantom also includes support for implicit declarations, dynamically-sized lists, and general purpose higher-order functions. These features are modelled on their counterparts in other interpreted languages (such as Scheme [IEE91] and Python [Ros92]).

3.1 Object Model

Phantom supports object-oriented programming, in a manner similar to Modula-3. Phantom objects have *attributes* (containing state information), a number of *methods* (for performing operations), and support single-inheritance. Phantom uses a class-based object model (rather than prototypes, as in Obliq [Car95] or Self [US87]). While classes are more verbose than prototypes, we feel that classes provide for a cleaner separation between the interface and implementation of objects, and scale better for large applications.

Objects are the focus of communication in Phantom. A Phantom program will generally make its services available to other programs by registering object values with a *name service* – a Phantom application server which maps string names to network addresses[2] of Phantom objects.

Objects (and lists, which are a kind of object) are the only values which are passed by reference. This has an important property when passing object values to Phantom programs at remote sites: objects are never implicitly migrated to remote sites as the result of an assignment, procedure call or return statement. Instead, the object remains stationary and a network reference is passed to the remote site. If migration of objects across sites is required, it must be performed explicitly by the programmer. While this violates location transparency to some degree, we feel that only the programmer can make reasonable decisions about when and where to migrate objects.

3.2 Distribution Model

The distribution model of Phantom borrows heavily from the distributed lexical scoping semantics of Obliq [Car95]. The basic concepts are illustrated in figure 2.

[2]These network addresses take the form of *global location addresses*, described in section 3.2.

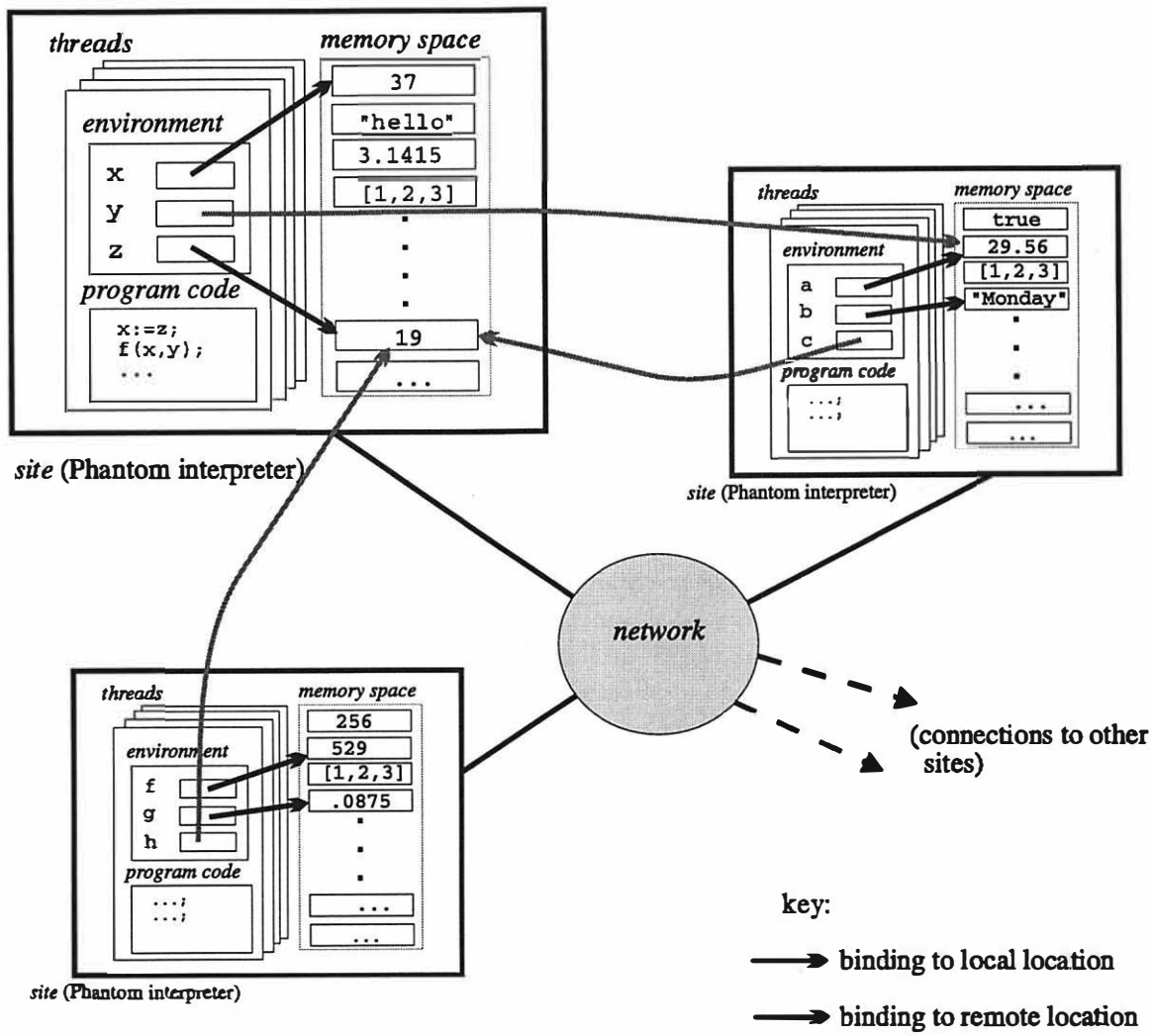


Figure 2: Phantom network architecture.

A *network* connects a number of *sites*. A *site* is an invocation of the Phantom interpreter on some host machine, and has a *site address* which uniquely identifies that site throughout the network. In the current implementation, a site address is simply a pair consisting of the IP address and port number of a TCP socket owned by the interpreter process. Note that a host running a multi-tasking operating systems may contain several sites (corresponding to multiple invocations of the Phantom interpreter).

Within a site, the interpreter maintains a single *memory space* for the program it is executing. This memory space contains a number of *locations*. Each location has a *location address*, which uniquely identifies the location within the interpreter's memory space, and holds a *value*.

Each Phantom program executes as a number of *threads* within the interpreter. Threads are provided through a library which implements the POSIX pthreads specification [IEE92]. Two data structures are associated with each thread:

1. A representation of the program code which the thread is executing. In the current implementation, the interpreter uses a sequence of byte codes for a virtual stack machine for this purpose.

2. An *environment*, which maps every variable or constant identifier appearing in the Phantom program to a *global location address*. A *global location address* is a pair consisting of a site address and a location address within the memory space of that site.

The Phantom interpreter uses environments to provide transparent distribution for Phantom programs. Each statement in a Phantom program may make reference to constant and variable identifiers. As the interpreter executes a statement, it uses the program's current environment to map these identifiers to their corresponding global location addresses. The interpreter then performs the appropriate operation on each location according to the defined semantics of the language. If the global location address refers to a location within the local interpreter's memory space, the operation is performed directly by the interpreter. If, however, the global location address refers to a location in the memory space of another site, the interpreter sends a request to the remote site asking it to perform the given operation.

3.3 Example Application: Generic Client and "Hello" Server

The following example illustrates how the interpreter and runtime provide Phantom's distributed semantics, and also illustrate the basic techniques for developing dynamically extsenible, distributed, interactive applications in Phantom. There are two programs presented. The first is a generic client program (such as might be launched as an "external viewer" from a WWW browser), and the second is a specific application server with which the client can communicate.

The generic client uses the information in a Uniform Resource Locator (URL) [BL94] to obtain a reference to a remote application server. Once the client has obtained a reference to the application server, it obtains an autonomous *agent* from the sever: a procedure received from the remote site which the client executes locally.

The "Hello" Server is as an example of a specific application server. It accepts requests from clients and returns to each client an agent which creates an instance of an interactive "hello world" object at the client's site.

The general purpose client could be invoked using a command line such as:

```
$ phi AppClient phi://server.host.name/HelloServer
```

which starts the Phantom interpreter (`phi`) executing the module `AppClient` (the name of the client program) with the URL for the Hello server as a command-line argument available to the client.

The agent obtained from the Hello server creates a window on the client's display as shown in figure 3:

Figure 3: Window Created on Client's Display by Hello Server

All user interface events for this window are handled by code for the agent, which is executed at the client site. When the user presses the button labelled "Hello", the agent responds by printing the message `''Hello, World''` to its output stream. When the user presses the button labelled "Quit", the agent returns control to the client program, which then exits.

While this is a simple example, it serves to illustrate most of the important features of the application domain for which Phantom was designed. This example is both distributed and interactive, the client is dynamically extensible, and all interactive user interface events are handled at the client site. These same principles apply to other, more sophisticated applications in the target domain.

3.3.1 Application Server Interface

An application server makes its services available to clients by registering an instance of an `AppServer.T` object with a name server. The `AppServer.T` type is implemented by *every* specific application server, and is also known to the generic client. This shared interface is as follows:

```
interface AppServer;

import rd, wr;

(* An Agent is simply a procedure executed at the client site *)
type Agent = proc (istrm: rd.T; ostrm: wr.T);

(* AppServer.T -- an application server *)
type T=object (serialised, protected)
methods
  (* generate a new agent for execution at the client *)
  generate_agent(): Agent perm x;
end;

end AppServer.
```

This interface defines two types: `AppServer.Agent` (describing the type of the agent given to the client for local execution), and `AppServer.T`, the abstract type of an application server. The generic client obtains an agent from an application server by first obtaining a reference to an `AppServer.T` (using the name service), and then invoking its `generate_agent()` method. As will be shown later, specific application servers are implemented by creating subtypes of `AppServer.T`.

The type `Agent` makes use of the fact that procedures are first-class types in Phantom: procedures may be assigned to variables, passed as parameters, and returned from procedures, just like values of other fundamental language types. In this example, an **`Agent`** is a procedure taking two parameters (which must be supplied by the client). Such parameters represent the *services* which an *execution site* (i.e. the client) makes available to agents it receives from across the network. In a more general interface, such services would encapsulate all of the local resources the execution site is willing to provide to the agent: a local audio service, a 3-D rendering service, etc. For simplicity, the only services provided to agents in this example are input and output streams for reading and writing messages.

The object type, `T`, has two *qualifiers* (the **`serialised`** and **`protected`** keyword qualifiers[3]), no attributes, and a single method: `generate_agent()`.

[3] For those familiar with Obliq, both of the `serialised` and `protected` qualifiers have the same semantics as their counterparts in Obliq.

The *qualifiers* give the object type special semantics. A `serialised` object type ensures that only one external method invocation or attribute update is active at a time in the presence of multiple concurrent requests. The `protected` qualifier prevents any attributes of the object from being updated externally, and prevents the object from being copied. The `protected` qualifier is of little immediate use on the object type defined here (since it has no attributes), but the qualifier is part of the type, and will hence be carried down into subtypes of `T`.

Ownership and Access Control

The `generate_agent()` method of type `T` has a name, a procedure signature, and a *permissions specification* (given by `perm x` following the signature). Permissions specifications are used to set the *access control* properties of attributes and methods.

Each Phantom object is stored in the memory space of an interpreter, which communicates with other interpreters across a network. Each object has an *owner*, which is represented at runtime as a `sys.user` object corresponding to the user who started the interpreter containing the object.

The permissions specification specifies what operations on the object may be performed by users other than the owner. An operation on an object by a user other than the owner happens when the interpreter receives a request from a Phantom program running on a different site, and is discussed in detail in section 3.5.2.

Each permissions specification consists of a bit-mask of zero or more of the three *permission bits* `r`, `w` and `x`, corresponding to *read*, *write* and *execute* permission, respectively.[4] If no permissions specification is given for an attribute or method, the default is that all permission bits are turned off.

In the current example, the `generate_agent()` method of `AppServer.T` has its `x` bit set in the permissions specification to allow clients at remote sites to invoke this method.

3.3.2 Client Program

The client program is straightforward: it obtains the global location address of an `AppServer.T` object from the name service (defined in the interface `ns`) using the application's URL, invokes the `generate_agent()` method of the server to obtain an agent, and executes the agent locally, supplying the standard input and output streams of the client as the parameters to the agent. The code for the client is as follows:

```
(* AppClient.pm -- implementation of a general-purpose network client *)
module AppClient;

import AppServer, Tk, stdio, ns, sys, urllib;

const
  urlstring = "phi://server.host.name/HelloServer";

begin
  try
    url:=urllib.parse(urlstring);
    name_server:=ns.find(url.host);
```

[4] Note that the `r` and `w` bits apply only to attributes, and the `x` bit applies only to methods.

```
    app_ref:=name_server.lookup(url.path);
    app_server:=narrow(app_ref, AppServer.T);

    (* obtain the agent from the server *)
    local_agent:=app_server.generate_agent();
    (* and execute the agent locally *)
    local_agent(stdio.stdin,stdio.stdout);
  except
    urllib.malformed =>
         stdio.stderr.puts("error: malformed URL: " @ urlstring @ "\n");
  | sys.narrow_failure =>
         stdio.stderr.puts("error: URL does not refer to an AppServer\n");
  | ns.not_available =>
          stdio.stderr.puts("error: could not contact name server at host " @
                            url.host @ "\n");
  | ns.unknown_service =>
          stdio.stderr.puts("error: application " @ url.path @
                            " not registered with nameserver.\n");
  end;

end AppClient.
```

The first item to note in the above example is that there are no explicit variable declarations. In Phantom, it is not necessary to declare a local variable prior to its use in a statement block. The first assignment to an undeclared identifier will declare that variable in the local scope, with a type derived from the expression on the right-hand side of the assignment statement[5]. All subsequent references to the identifier in the statement block will be type-checked against this automatically derived type.

The client program works as follows: First, the program calls `urllib.parse()` to parse the URL given by `urlstring`. (In this example, `urlstring` is given as a constant; in practice it would use a command-line argument.) The procedure `urllib.parse()` returns the URL as a record with seperate `protocol`, `host` and `path` fields. Next, the client attempts to contact a Phantom name sever running on the host given in the URL, using the procedure `ns.find()`. The `ns` module and interface is part of the standard Phantom library, and locates a name server object on a local or remote site using a well known TCP port. The variable identifer `name_server` is assigned the global location address of the name server object, returned from the call to `ns.find()`. Any subsequent operation on the identifier `name_server` is forwarded transparently by the interpreter to the name server object, which performs the operation and returns the result.

Next, the `lookup()` method is invoked on the `name_server` object to obtain a reference to the application server, using the pathname part of the URL (`"HelloServer"` in this example). The `lookup()` method returns its result as type `any`; the statement following the lookup performs a type-safe runtime type conversion using `narrow()` to convert this value to an object reference of the appropriate type. Note that after the client performs the `lookup()` operation, the name server is no longer involved in the communication between the client and the server; it just

[5]Note that explicit variable declarations may still be provided (since they increase readability for large programs), in which case the type given in the explicit declaration is used for type checking.

provides a mechanism for bootstrapping the connection between them. Once the client has a reference to the remote `AppServer.T` object, operations can be performed on the object reference in the same manner as with the `name_server` object; the runtime handles any network communication required.

Next, the client invokes the `generate_agent()` method of the `app_server` to obtain an agent for local execution. Since the value returned from `generate_agent()` is a procedure, this will result in obtaining the code for the procedure from the application server. This code will be dynamically loaded into the memory space of the client, and the variable `local_agent` will refer to the *closure* for this procedure. As will be discussed later, the semantics of transmitting code across sites ensures that this is a safe operation: code received from a remote site and executed locally can not gain unauthorised access to any local resources.

Finally, the client invokes `local_agent`, passing as parameters the standard input and output streams of the client program. Thus, the client has no information about specific applications hard coded into it, but dynamically obtains application-specific behaviour by receiving code from the server. This generic client program could be used without modification as a client for any application-specific server.

It is also worth noting that the client wraps the entire body of its mainline in a `try-except` statement, to catch some of the exceptions which may be raised in the process of obtaining the application-specific agent, and reports these as errors to the user. More sophisticated error recovery mechanisms could be implemented using this facility.

3.3.3 Server Program

The application-specific server is a "Hello, World" server. It returns to clients an agent which, when executed at the client site, creates a graphical, interactive "Hello, World" window on the client's display. The agent uses the library interface between Phantom and the Tk toolkit to implement the graphical user interface for the agent.

The server defines the type `ServerImpl` as a subtype of `AppServer.T`. This is a common technique in Phantom: an object type appearing in an interface will describe the external view presented to clients, and a subtype will be used to implement the application-specific server. The source code for the server program is as follows:

```
module HelloServer;

import AppServer, Tk, rd, wr, ns, stdio;

(* ServerImpl is the type of the "hello" application server; implemented as a
 * subtype of AppServer.T
 *)
type ServerImpl=AppServer.T object
end;

(* Hello is the object type instantiated at the client site *)
type Hello=Tk.Frame object
   quit: Tk.Button;
   msg: Tk.Button;
   wstrm: wr.T;      (* stream on which to write messages *)
methods
```

```
   CreateWidgets();
   say_hi();
end;

(* methods of Hello: *)
proc Hello.CreateWidgets(self: Hello)
begin
  self.quit:=new(Tk.Button,
                 master:=self,
                 text:="Quit",
                 fg:="red",
                 command:=lambda () { self.exit(); });
  self.quit.pack(side:=Tk.left);
  self.msg:=new(Tk.Button,
                master:=self,
                text:="Hello",
                command:=lambda () { self.say_hi(); });
   self.msg.pack(side:=Tk.left);
end;

(* init() method -- called automatically to initialise new instances *)
proc Hello.init(self: Hello)
begin
  Tk.Frame.init(self); (* call super-class init method *)
  self.pack();
  self.CreateWidgets();
end;

proc Hello.say_hi(self: Hello)
begin
  self.wstrm.puts("Hello, world!\n");
end;

(* methods of Hello Server: *)
proc ServerImpl.generate_agent(self: ServerImpl): AppServer.Agent
 (* client_agent() is the procedure returned by generate_agent() and
  * executed at the client site
  *)
 proc client_agent(istrm: rd.T; ostrm: wr.T)
 begin
   hello_app:=new(Hello,
                  wstrm:=ostrm);
   hello_app.main_loop();
 end;
begin
  return client_agent;
end;
```

```
begin
  (* create an instance of the server, and register it with the local name
   * service
   *)
  hello_server:=new(ServerImpl);
  name_server:=ns.find();
  name_server.register("HelloServer",hello_server);
end HelloServer.
```

The server is implemented as follows:

First, the actual server object is implemented as a subtype of the object type `AppServer.T`. The subtype (`ServerImpl`) does not add any attributes or methods to `AppServer.T`, it simply overrides the `generate_agent()` method of `AppServer.T`. Hence, the body of the `ServerImpl` is empty, since it does not have any specific attributes or methods, and, in Phantom, method overrides are not stated explicitly in the object type.

Next, the application server defines the object type `Hello` as a subtype of `Tk.Frame`. No instance of this type is ever created at the server site; instead, an instance of this type is created at the client site by the application-specific agent. When the agent is transmitted from the server to the client, all information about types used within the agent is transmitted across the network and reconstructed at the client site. For object types, this includes both the information necessary to construct instances of the type, and the code for any methods. Note that a type may refer to other types in its definition, and types may be recursive; the runtime will transmit all necessary type information, including information about types referenced indirectly or recursively.

The agent returned to clients is the procedure `client_agent()` defined in the `generate_agent()` method of `ServerImpl`. The `client_agent()` procedure (executed at the client site) creates a new instance of type `Hello` at the client site, and invokes the `main_loop()` method of `Hello` to process GUI events which happen in this object. The `main_loop()` method of `Hello` is inherited from `Tk.Frame`, the parent type of `Hello`.

Finally, the mainline of `HelloServer` creates a new instance of `ServerImpl`, and registers this with the local name server. When the Phantom interpreter is invoked to run the server application, it would be invoked with the `-noexit` option, to ensure that the interpreter does not exit after initialisation, but instead waits idly for requests from remote sites.

3.4 Semantics of Procedure Transmission

The previous example makes use of the fact that procedures are first class types in Phantom. This has an important property when passing parameters to (or returning values from) methods of remote objects. If a procedure value is passed as a parameter or returned as a return value, a *closure* for the procedure is transmitted across the network.

Phantom programs are lexically scoped: that is, the location to which a free identifier is bound is purely a static function of where the procedure is defined, and is not a dynamic function of the procedure call stack. This property is used to give higher-order procedures an intuitive and secure meaning in a distributed context. These semantics are illustrated in figure 4:

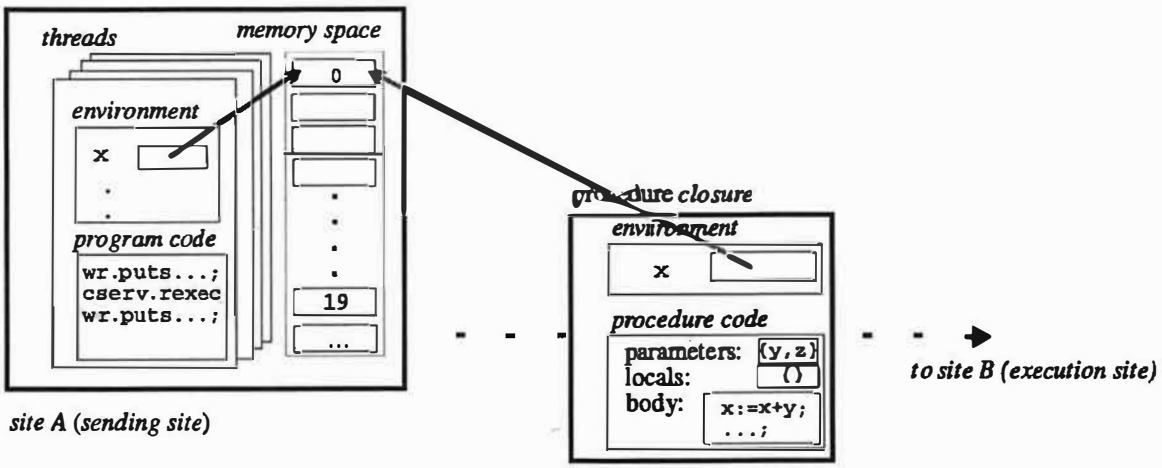


Figure 4: Transmission of closures across sites.

When a procedure value is sent to a remote site, the local interpreter sends the procedure as a *closure*. The closure contains two pieces of information:

1. A representation of the code in the body of the procedure. This could be either a direct representation as source code text, or some internal representation such as a linearised parse-tree or byte-codes for a virtual machine.

2. An *environment*, which maps all free variable identifiers which appear in the procedure to global location addresses.

Transmitting the set of bindings along with the code for the procedure preserves the correct lexical scoping semantics when the procedure is executed at the remote site. When the procedure body makes reference to a free identifier, the binding to the global location address ensures that the operation is performed on the location where the identifier was bound originally.

The "Hello, world" example illustrates a limited case of transmitting procedures across sites. In that example, the procedure which is transmitted to the client site as the application-specific agent has no free variable identifiers. That is, the procedure `client_agent()` does not refer to any variables from its enclosing scope. This is an example of a "disconnected" agent: all information needed to execute the procedure at the client site can be encapsulated in the code of the closure, and the closure's environment will be empty. Although `client_agent()` does refer to *types* (such as the `Hello` object type[6]) from enclosing scopes, this type information is transmitted to the client site as part of the code of `client_agent()`'s closure.

If the body of `client_agent()` made reference to a variable in its surrounding scope, the environment of the closure transmitted to the client would contain a binding to the location of the variable at the server site. This would have the effect of creating a "connected" agent: one which carries its network connections with it. This facility could be used, for example, to create a distributed multiplayer game. The agent transmitted to the client could simply invoke operations on an object (representing the opposing player) declared in one of the agent's enclosing scopes. Any time the agent (executing at the client site) performed such an operation, the

[6]Note that the methods of object types are themselves procedures. When the information about an object type is sent across sites, the methods of the object type are transmitted as true closures in the same way as other procedures.

client runtime would use the environment transmitted with the agent to forward the operation to the site where the object resides.

3.5 Security Considerations

Phantom's distribution model raises a number of interesting security issues. The most important issue is that raised by the ability to send code across sites: the language and runtime must provide strong guarantees about the safety of executing code received from a potentially untrustworthy server. Phantom addresses this issue, and also provides two other forms of security support: secure user authentication and access control, and unforgeable global location addresses.

3.5.1 Security of Code Transmission

The principle concern when transmitting code across sites is security. The language and runtime environment must be able to guarantee that program code which is received from a remote site and executed locally will not have access to any local resources which could not have been accessed via RPC from the remote site.

Phantom makes this guarantee through adherence to lexical scoping in the context of distribution and higher-order functions. In practical terms, the implementation guarantees lexical scoping by passing a set of bindings for all free identifiers along with the code for a procedure. When an interpreter receives a procedure from a remote site, it can perform a single, static check to ensure that all free identifiers in the code for the procedure have a corresponding entry in the set of bindings received with the procedure. If any free identifier does not have a corresponding binding, the interpreter will abort the operation requested by the remote site and return a security violation exception.

The language has no general purpose pointer types, which is a crucial aspect of this approach to security. Eliminating general purpose pointers ensures that the only way for a procedure to refer to resources outside the body of the procedure is through free identifiers. Because the implementation ensures that free identifiers are handled through strict lexical scoping, executing procedures received from remote sites is guaranteed to be secure: there is simply no mechanism for the procedure to gain unauthorised access to any local resources.

3.5.2 Authentication and User Identity

In Phantom, the identity of every user has a runtime representation as a `sys.user` object. Procedures in the standard interface `sys` provide access to a `sys.user` object which represents the *current effective user indentity*. A simple, secure authentication protocol (based on public key encryption) is used when the connection between two Phantom interpreters is first established. The secure authentication protocol ensures that an interpreter can obtain an unforgeable `sys.user` object representing the user who started execution of the interpreter at the remote site. This object is used as the current effective user identity when the local interpreter executes any methods or procedures as the result of a request from the remote interpreter.

As discussed in section 3.1, every attribute or method of an object has a set of access control bits which determine the operations that may be performed on the object by remote users. Access checks performed by the interpreter are always based on the current effective user identity. An object operation will succeed if either the user running the current thread is the owner of the object (which is always true for purely local operations), or if the permission bits permit relevant access to the attribute or method for other users.

The security model provided at the *language level* by Phantom is minimal by design. The goal of the language-level primitives is to provide a simple, secure mechanism for partitioning an application into parts which are and are not network-accessible. For large-scale applications, which have more sophisticated security requirements, we plan to augment the language-level primitives with a library that provides complex principals, access control lists and encryption, in a manner similar to Taos [WABL93].

3.5.3 Global Location Addresses and Security

A global location address is the handle used by an interpreter to access locations stored in the memory space of remote sites. As described in section 3.2, a global location address consists of a site address (of the interpreter which owns the location), and a location address (relative to the memory space of that site).

The relative location address part of a global location address is a 128-bit key generated by the site which owns the location, rather than just a pointer into the interpreter's memory space. Because this key space is large and sparsely populated, the key acts as a software *capability* [Fab74] for locations in the interpreter's memory space. Thus, private data can be safely stored in ordinary Phantom program variables and shared with authorised users, without fear of a rogue site gaining access to the data by guessing the location's address.

4 Current Status

At the time of writing, the design of the language and interpreter are complete. The interpreter has been implemented, and supports all of the features in the language core, including static typing, type-safe implicit declarations, objects, interfaces, threads, exceptions, garbage collection, dynamically sized lists and higher-order functions, and includes a library interface to the Tk toolkit. A number of small demonstration programs have been written in Phantom. The initial results are limited but encouraging: the language's Modula-3 heritage affords it a number of powerful features, while still maintaining overall coherence and simplicity.

The implementation of the networking subsystem in the interpreter required for distribution is not yet complete; we expect to complete that part of the interpreter over Summer, 1995.

5 Future Work

Once the distribution facilities in the interpreter are available, we intend to use Phantom to develop a number of novel distributed applications. The most compelling of these ideas is a graphical, extensible distributed conferencing system. This will involve implementing a distributed version of LambdaMOO [Cur92] in Phantom. LambdaMOO is a "text-based virtual reality" which uses its own interpreted, object-oriented language to provide a general-purpose, dynamically extensible conferencing environment. However, LambdaMOO is totally centralised, and there are fundamental security problems in extending LambdaMOO (in its present form) to a distributed implementation. We are confident that Phantom will make it possible to implement a distributed conferencing system with all the flexibility and power of LambdaMOO, but with support for a graphical user interface, strong network security guarantees, and a distribution model which scales to support more users.

6 Acknowledgements

Special thanks to David Abrahamson, Luca Cardelli, Dan Connolly, Bill Janssen, Danny Keogan, Ciaran McHale, Killian Murphy and Brendan Tangney, who read early drafts of the language report and this paper, and provided valuable feedback on the exposition and language design.

References

[BL94] Tim Berners-Lee. Internet RFC 1738: Uniform Resource Locators (URL). URL: http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html, December 1994.

[BLC93] Tim Berners-Lee and Daniel W. Connolly. Hypertext Markup Language: A Representation of Textual Information and Metainformation for Retrieval and Interchange. URL: http://info.cern.ch/hypertext/WWW/MarkUp/HTML.html, 1993.

[BLCGP92] Tim Berners-Lee, R. Cailliau, J-F Groff, and B. Pollermann. World-Wide Web: The Information Universe. *Electronic Networking: Research, Applications and Policy*, 2(1):52–58, Spring 1992.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3), September 1989.

[Car95] Luca Cardelli. A Language with Distributed Scope. In *Principles of Programming Languages*, January 1995. URL: http://www.research.digital.com/SRC/Obliq/Obliq.html.

[Cur92] Pavel Curtis. LambdaMOO Programmer's Manual. Technical report, Xerox Corporation, Palo Alto Research Centre, October 1992. URL: ftp://parcftp.xerox.com/ProgrammersManual.ps.

[Fab74] Robert S. Fabry. Capability-Based Addressing. *Communications of the ACM*, 17:403–412, July 1974.

[Gro92] The Object Management Group. The Common Object Request Broker: Architecture and Specification, 1992.

[IEE91] IEEE. *Std. 1178-1990. IEEE Standard for the Scheme Programming Language.* Institute of Electrical and Electronic Engineers, 1991.

[IEE92] IEEE. *P1003.4a/D6 Threads Extension for Portable Operating Systems (Draft 6).* Institute of Electrical and Electronic Engineers, February 1992.

[JSS94] Bill Janssen, Mike Spreitzer, and Denis Severson. Inter-Language Unification, 1.6.4. Technical Report P94-00058, Xerox Corporation, Palo Alto Research Centre, May 1994. URL: http://ftp.parc.xerox.com/pub/ilu/ilu.html.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[RJN94] David W. Robertson, William Johnston, and Wing Nip. Virtual Frog Dissection: Interactive 3D Graphics via the Web. In *Proceedings of the Second International WWW Conference*, 1994. URL: http://george.lbl.gov/ITG.hm.pg.docs/dissect/info.html.

[Ros92] Guido Van Rossum. Python Language Reference Manual. software documentation, 1992.

[RT74] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[SG90] James W. Stamos and David K. Gifford. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering*, 16(7), July 1990.

[Str92] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1992.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *ACM Sigplan Notices*, 2(12), 1987.

[WABL93] Edward Wobber, Martin Abadi, Mike Burrows, and Butler Lampson. Authentication in the Taos Operating System. Technical Report 117, Digital Equipment Corportation, Systems Research Centre, Decemeber 1993.

A Framework for Higher-Order Functions in C++

Konstantin Läufer

Loyola University of Chicago
laufer@math.luc.edu

Abstract

C and C++ allow passing functions as arguments to other functions in the form of function pointers. However, since function pointers can refer only to existing functions declared at global or file scope, these function arguments cannot capture local environments. This leads to the common misconception that C and C++ do not support function closures.

In fact, function closures can be modeled directly in C++ by enclosing a function inside an object such that the local environment is captured by data members of the object. This idiom is described in advanced C++ texts and is used, for example, to implement callbacks.

The purpose of this paper is twofold: First, we demonstrate how this idiom can be generalized to a type-safe framework of C++ class templates for higher-order functions that support composition and partial application. Second, we explore the expressiveness of the framework and compare it with that of existing functional programming languages.

We illustrate by means of various examples that object-oriented and functional idioms can coexist productively and can be used to enhance the functionality of common classes, for example, of nonlinear collections such as trees. A C++ implementation of the framework is available on request.

1 Introduction

The programming languages C [HS87] and C++ [ES90] allow passing functions as arguments to other functions in the form of function pointers. However, since function pointers can refer only to existing functions declared at global or file scope, these function arguments cannot capture local environments. This leads to the common misconception that C and C++ do not support function closures.

On the contrary, function closures can be modeled in C++ by enclosing a function inside an object such that the local environment or parts thereof are captured by data members of the object. This is possible because objects in C++ are essentially higher-order records, that is, records with fields that can contain not only values, but also functions [Red95].

The idiom of objects that enclose functions is first described by Coplien [Cop92], who calls these objects "functors", and further developed by Kühne [Küh95], who calls them "function objects". In this paper, we use the term "functoid" for brevity and to avoid confusion with established meanings of "functor" in other areas of computer science. The functoid idiom has been used, for example, in callbacks and iterators with type-safe interfaces. A related idiom that focuses on maintaining a binding between a receiver and a function is called "command" by Gamma et al. [GHJV93]. Unlike Kühne, Gamma et al. do not establish a relationship between the "command" and "iterator" idioms. The "command" idiom is also known as "action" or "transaction" and is used in various object-oriented application frameworks [JF88], including ET++ [WGM88], InterViews [LCI+92], MacApp [App89], and Unidraw [VL90].

Nevertheless, various existing class libraries provide internal (passive) iterators with weakly-typed interfaces and place the responsibility of applying suitable type casts on the user. For example, Borland C++ [Bor94] uses the following internal iterator in its container class templates.

```
void Container<Item>::forEach
    (void(* f )(Item&, void*), void* args);
```

The purpose of the second argument of `f` and the argument `args` is to allow passing specific arguments to `f`. Using the functoid idiom, the iterator could be provided in the following type-safe way.

```
void Container<Item>::forEach
    (Visitor<Item>& f);

template <class Item> class Visitor
{
public:
    virtual void operator()(const Item&)
    = 0;
};
```

By deriving from the class template `Visitor`, information can be passed to and from the iterator in a type-safe manner, where specific arguments are passed as arguments to the constructor of the visitor. For example, the following visitor adds the elements of a container of integers.

```
class Adder : public Visitor<int>
```

```
{
public:
    Adder(int& s) : sum(s) { }
    virtual void operator()
        (const int& item)
    { sum += item; }
private:
    int& sum;
};

int sum = 0;
Adder add(sum);
container.forEach(add);
cout << sum;
```

The purpose of this paper is twofold: First, we demonstrate how the functoid idiom can be generalized to a type-safe framework of C++ class templates for higher-order functions that support composition and partial application. The framework could be translated to other object-oriented languages that support inheritance and genericity. Second, we explore the expressiveness of the framework and compare it with that of existing functional programming languages.

We show informally that there is a simple compositional translation from functional programs to the framework. We illustrate by means of various examples that object-oriented and functional idioms can coexist productively and can be used to enhance the functionality of common classes, for example, of nonlinear collections such as trees. To integrate the framework with C and C++ programs, we incorporate an existing mechanism to convert member functions back to nonmember functions. A C++ implementation of the framework is available on request.

In the remainder of this paper, Section 2 describes in detail the requirements, the implementation, and the structure of the functoid framework. Section 3 conducts a case study in which a typical functional program expressed within the framework. Section 4 explains how an existing conversion mechanism from C++ member functions to ordinary nonmember functions is incorporated in the framework. Section 5 concludes with an assessment of this work and a look at related and future work.

2 Functoids: An Abstraction of Functions

This section introduces the framework of functoids. In this framework, a *functoid* is an abstraction of the familiar concept "function".

Requirements

We first establish the requirements of the functoid abstraction. Our goal is to provide a *type-safe* abstraction, that is, there should be no need for type casts or untyped pointers at the user level. The abstraction should be provided in the form of C++ classes or class templates. We require that the abstraction supports the following essential operations performed on or by functions:

- *Application*

  Functoids can be applied to arguments. When a functoid is invoked by applying it to one or more arguments of appropriate types, the functoid returns a value of the appropriate result type.

- *Creation*

  Functoids can be created statically or dynamically. Upon creation, the functoid can capture and remember parts of the current environment.

- *Composition*

  Functoids can be composed with one another. When a functoid `f` is composed with another functoid `g`, the result of the composition is a new functoid `h`. When `h` is applied to an argument, it first obtains an intermediate result by applying `g` to the argument and then returns as a final result the application of `f` to the intermediate result.

- *Partial application*

  Functoids can be applied partially to fewer arguments than they actually accept. The result of partial application is a new functoid that accepts the remaining arguments. The conversion to a functoid that takes its arguments one at a time is known as "currying" in functional programming terminology.

- *Conversion*

  Functoids are equivalent to ordinary functions. An ordinary function can be converted to a functoid, and a functoid can be converted to an ordinary nonmember function when such a function is required, for example, as a callback function for an existing library. Conversion back to nonmember functions is difficult and will be addressed in Section 4.

- *Extension*

  Functoids provide extensible functionality. We can add application-specific operations to a

functoid besides the basic operations described above.

We implement our abstraction as a class template Fun that provides the interface to the abstraction *functoid* and is parameterized by the types of argument and result. The creation requirement will be handled by the constructors of this class, and the application requirement is captured by a function call operator of the appropriate type.

```
Out Fun<In,Out>::operator()(In arg) const;
```

The question arises how users of the functoid framework should incorporate their own functoid classes. The idea is that users derive concrete functoid classes from the class Fun, providing their own implementations of the function call operator. To make this approach work, the function call operator would have to be declared as virtual so that dynamic method selection is used, and functoids would always have to be passed and returned by pointer or reference [ES90]. On the other hand, memory management becomes an important issue when objects are not returned by value [Mey92]. What we want here is both call-by-value and dynamic method selection.

Fortunately, the *envelope/letter* idiom [Cop92], also known as the *bridge* pattern [GHJV93], gives us a way out of this dilemma. We apply this idiom to the framework as follows. We provide a class template called Fun to capture the interface of our abstraction. This is the envelope class, and functoids are passed and returned by value as instances of this class. We also provide an abstract class template called FunImpl for implementations of functoids. This class is an abstract letter class, and users of the framework provide their own functoid implementations by deriving from this class. The envelope class has a pointer to the letter class, and invocations of the function call operator in an envelope object are simply passed on to the letter object.

We first present the envelope class template Fun because it comes first conceptually, although it depends on the class template FunImpl. We provide two constructors, one to create a functoid from an existing functoid implementation, and a copy constructor that makes an explicit copy of the implementation of the functoid it copies. This is necessary so that no two functoid implementations are shared and the destructor can safely delete the corresponding implementation. Furthermore, we provide a function call operator that simply passes the function call on to the implementation, accessible via the pointer impl.

```
template <class In, class Out> class Fun
{
public:
    Fun(FunImpl<In,Out>* const f)
        : impl(f) { }
    Fun(const Fun& fun)
        : impl(fun.impl->copy()) { }
    ~Fun() ( delete impl; )
    Out operator()(In arg) const
    { return (*impl)(arg); }
private:
    FunImpl<In,Out>* const impl;
};
```

We now present the abstract letter class template FunImpl. It has a virtual destructor so that the appropriate destructor in a concrete subclass is invoked when a functoid deletes its implementation. Furthermore, it provides a member function that makes a copy of the receiver to support the copy constructor of the envelope class.

```
template <class In, class Out>
class FunImpl
{
public:
    virtual ~FunImpl() { }
    virtual Out operator()(In arg) const
    = 0;
    virtual FunImpl<In,Out>* copy() const
    = 0;
};
```

Composition

The next issue deals with the composition of functoids. In a functional programming language such as ML [MTH90], a function that composes two functions can be expressed as follows:

```
fun compose(f,g) = fn x => f(g(x))
```

The form "fn x => e" creates an anonymous function with argument x and body e. Thus the composition yields a new function with argument x and result f(g(x)). The composition is permitted only if the result type of g is compatible with the argument type of f. The new function has the same argument type as g and the same result type as f.

To avoid excess parameterization of the template Fun, we provide this functionality as a nonmember function that returns a functoid composed from the two functoid arguments. This resulting functoid is an instance of the class template Compose and holds the two functoids to be composed; the composition itself is carried out in the function call operator of this class. Both the class and the function templates have three type parameters for the argument, intermediate, and result types.

```
template <class In, class Med, class Out>
```

```
class Compose : public FunImpl<In,Out>
{
public:
    Compose(const Fun<Med,Out>& f,
            const Fun<In,Med>& g)
        : ffun(f), gfun(g) { }
    virtual Out operator()(In arg) const
    { return ffun(gfun(arg)); }
    virtual FunImpl<In,Out>* copy() const
    {
        return new Compose<In,Med,Out>
                   (ffun, gfun);
    }
private:
    const Fun<Med,Out> ffun;
    const Fun<In,Med> gfun;
};

template <class In, class Med, class Out>
Fun<In,Out> compose(const Fun<Med,Out>& f,
                    const Fun<In,Med>& g)
{
    return Fun<In,Out>
        (new Compose<In,Med,Out>(f, g));
}
```

Conversion from nonmember functions

The next requirement is conversion from an ordinary nonmember function to a functoid. The reverse direction is discussed below in Section 4. It is not hard to create a functoid from a nonmember function. Such a functoid can be implemented with a data member that points to the function and a function call operator that passes its argument on to the function pointer.

```
template <class In, class Out> class Global
    : public FunImpl<In,Out>
{
public:
    typedef Out(* FunPtr )(In);
    Global(FunPtr f) : theFun(f) { }
    virtual Out operator()(In arg) const
    { return theFun(arg); }
    virtual FunImpl<In,Out>* copy() const
    { return new Global<In,Out>(theFun); }
private:
    FunPtr theFun;
};
```

To enable automatic conversion from an ordinary function to a functoid, we add the following constructor to the class template `Fun`, where `FunPtr` is defined as in the class template `Global`.

```
Fun<In,Out>::Fun(FunPtr f)
    : impl(new Global<In,Out>(f)) { }
```

Partial application

Another issue is how to deal with functoids that take more than one argument. In ML, a function that partially applies a function of two arguments to the first argument can be written as follows:

```
fun apply(f,x) = fn y => f(x,y)
```

The result of the partial application of `f` to the first argument `x` is a new function with a single argument `y` and result `f` applied to `x` and `y`. The argument type of the new function is the type of the second argument of `f`, and its result type is the result type of `f`.

In the framework, the class template for functoids with multiple arguments would have to be parameterized by all argument types and the result type. Therefore the framework has to provide an envelope and a letter class for each number of arguments that could reasonably arise. If the maximum number of arguments is exceeded, a solution is to group several arguments in a single object. However, our partial application requirement can be satisfied only if the functoid accepts its arguments one-by-one. A better approach would thus be to automate the generation of the class templates depending on the maximum number of arguments desired in the application. The structure of the framework is sufficiently systematic to make this a feasible option.

We now illustrate partial evaluation for functoids of two arguments. First comes the abstract letter class `Fun2Impl`, followed by the corresponding envelope class `Fun2`. These classes differ from `FunImpl` and `Fun` in that they have two function call operators: one that takes two arguments instead of one, and one that takes a single argument and returns a new functoid. The second function call operator provides partial evaluation by applying the functoid to the first argument only.

```
template <class In1, class In2, class Out>
class Fun2Impl
{
public:
    virtual ~Fun2Impl() { }
    virtual Out operator()
        (In1 arg1, In2 arg2) const = 0;
    virtual Fun2Impl<In1,In2,Out>* copy()
    const = 0;
};

template <class In1, class In2, class Out>
class Fun2
{
public:
    typedef Out(* Fun2Ptr )(In1, In2);
    Fun2(Fun2Impl<In1,In2,Out>* const f)
        : impl(f) { }
    Fun2(Fun2Ptr f)
        : impl(new Global2<In1,In2,Out>(f))
    { }
    Fun2(const Fun2<In1,In2,Out>& fun)
```

```
        : impl(fun.impl->copy()) { }
    ~Fun2() { delete impl; }
    Out operator()(In1 arg1, In2 arg2)
    const
    { return (*impl)(arg1, arg2); }
    inline Fun<In2,Out> operator()
        (In1 arg1) const;
private:
    Fun2Impl<In1,In2,Out>* const impl;
};
```

To complete our implementation of partial evaluation, we must implement the function call operator that takes only one argument. This operator returns an instance of the class template `Apply21`, which keeps track of the *first* argument and the original functoid. When the function call operator of an instance of `Apply21` is invoked with the *second* argument, the operator simply applies the original functoid to both arguments.

```
template <class In1, class In2, class Out>
class Apply21 : public FunImpl<In2,Out>
{
public:
    Apply21(const Fun2<In1,In2,Out>& fun,
            const In1& arg1)
        : theFun(fun), theArg(arg1) { }
    virtual Out operator()(In2 arg2) const
    { return theFun(theArg, arg2); }
    virtual FunImpl<In2,Out>* copy() const
    {
          return new Apply21<In1,In2,Out>
              (theFun, theArg);
    }
private:
    const Fun2<In1,In2,Out> theFun;
    const In1 theArg;
};

template <class In1, class In2, class Out>
inline Fun<In2,Out> Fun2<In1,In2,Out>::
operator()(In1 arg1) const
{
    return Fun<In2,Out>
        (new Apply21<In1,In2,Out>
            (*this, arg1));
}
```

For additional flexibility in combining partial evaluation and composition, we also provide partial evaluation without application to any arguments. In ML, such a function is written as follows:

```
fun curry(f) = fn x => fn y => f(x,y)
```

This function converts its argument `f` to a new function that takes its arguments one after the other instead of both at the same time.

In the framework, the additional member function `curry1` in class `Fun2` converts a functoid of two arguments to a new functoid of the auxiliary class `Curry1`. The function call operator in the new functoid takes one argument (the first one) and returns a functoid that takes one argument (the second one) by invoking the partial function call operator in the original functoid on the first argument.

```
template <class In1, class In2, class Out>
class Curry1
    : public FunImpl<In1, Fun<In2,Out> >
{
public:
    Curry1(const Fun2<In1,In2,Out>& fun)
        : theFun(fun) { }
    virtual Fun<In2,Out> operator()
        (In1 arg1) const
    { return theFun(arg1); }
    virtual FunImpl<In2, Fun<In2,Out> >*
    copy() const
    {
        return new Curry1<In1,In2,Out>
            (theFun);
    }
private:
    const Fun2<In1,In2,Out> theFun;
};

template <class In1, class In2, class Out>
Fun<In1, Fun<In2,Out> >
Fun2<In1,In2,Out>::curry1() const
{
    return new Curry1<In1,In2,Out>(*this);
}
```

Adding methods to functoids

The last requirement addresses the extensibility of functoids. We will want to add application-specific member functions to the basic functionality provided by functoids. This can be done by deriving a class `UserFun` from the envelope class `Fun` and a class `UserImpl` from the abstract letter class `FunImpl`. Similarly to the function call operator, the additional member function `f` is implemented in `UserFun` as a wrapper that invokes the real one in `UserImpl`. We are facing a minor problem: not only is the pointer `impl` to the letter object private in class `Fun`, but it also is of class `FunImpl`, which does not have the new member function. We solve this problem by making `impl` protected in `Fun` and casting it to class `UserImpl` in the member function `f`. This cast is safe, since it is hidden from the user of the class.

The following example of a class `Cont` for continuations illustrates this requirement. Besides application to a consumer object of some other class `Consumer`, a continuation supports the method `done` to check whether the continuation has finished. We therefore make the envelope class `Cont` a subclass of `Fun` and the associated letter class `ContImpl` a subclass of `FunImpl`, each instantiat-

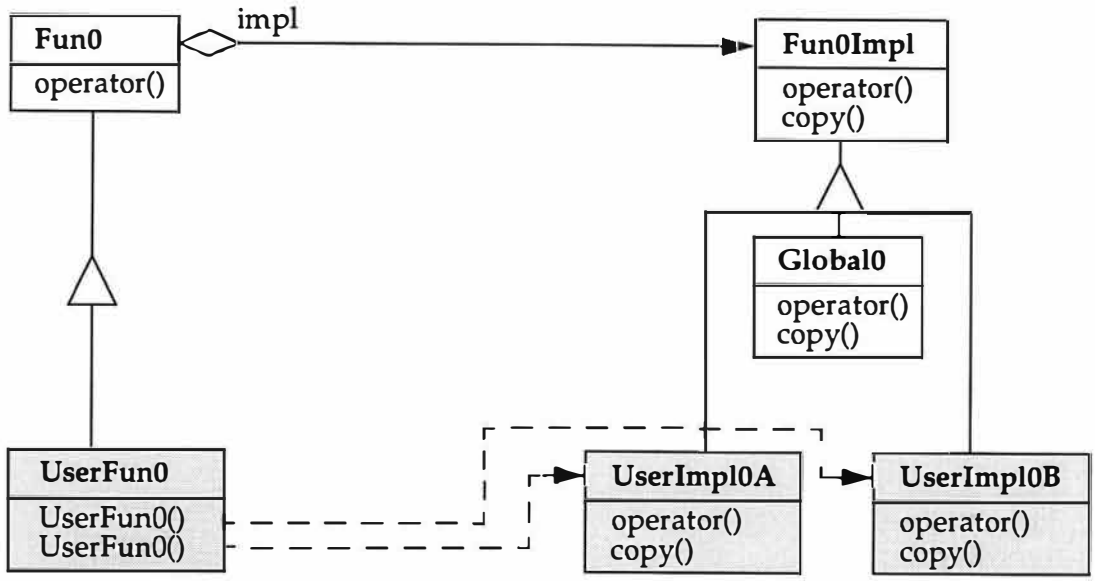


Figure 1: The functoid framework for zero arguments

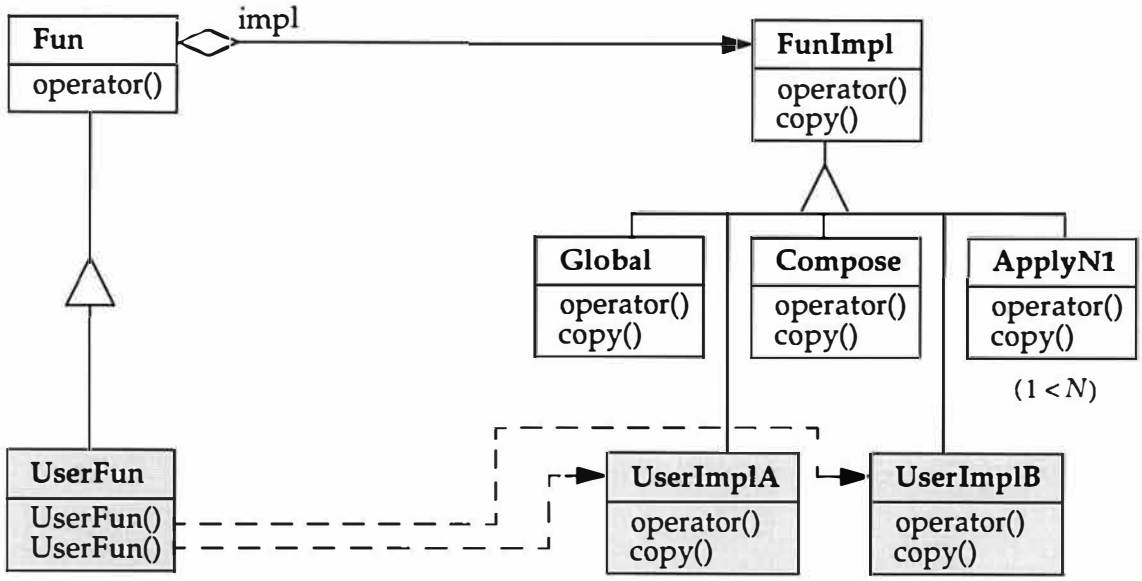


Figure 2: The functoid framework for one argument

ed with appropriate argument and result types. We extend the functionality of `Fun` and `FunImpl` by adding the member function `done` in the subclasses. The member function `done` in class `Cont` first casts the pointer `impl` to class `ContImpl` and then invokes the member function `done` in class `ContImpl`.

```
class ContImpl
    : public FunImpl<const Consumer&, bool>
{
public:
    virtual bool done() const { ... }
};
```

```
class Cont
    : public Fun<const Consumer&, bool>
{
public:
    bool done() const
    { return ((ContImpl*) impl)->done(); }
    ...
};
```

The structure of the functoid framework

Since the framework uses the envelope/letter idiom, it consists of separate abstraction and implementation class hierarchies. There are sub-frame-

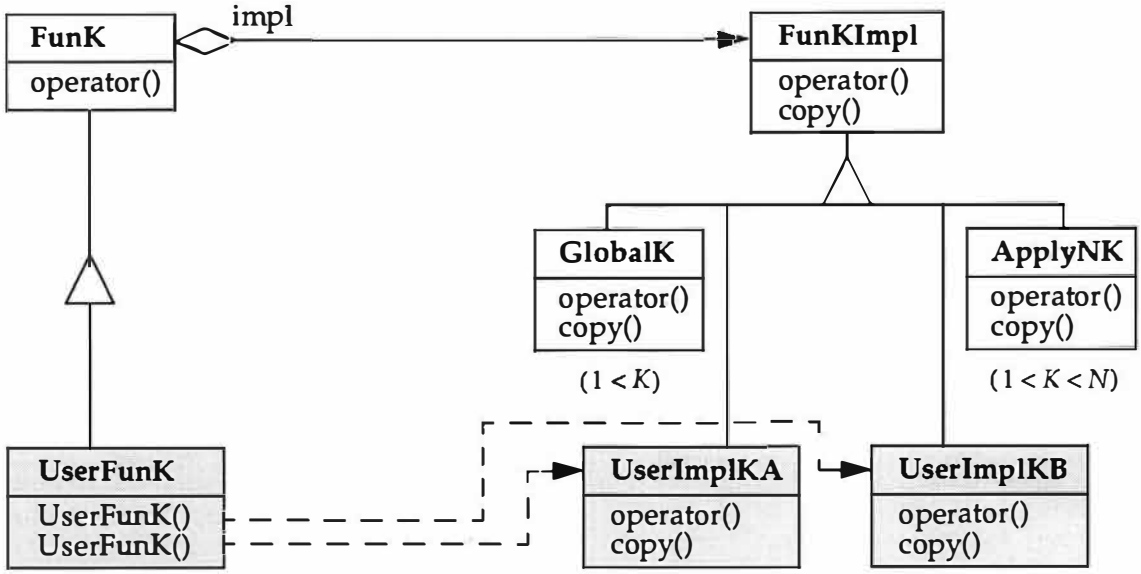


Figure 3: The functoid framework for two or more arguments

works for functoids of zero, one, and more arguments. We first describe the case of a single argument. The framework provides an abstraction class `Fun`, which is the class for functoids in user programs. The framework also provides an abstract implementation class `FunImpl`, from which users derive their own implementations of functoids by overriding the function call operator. Users of the framework derive classes `UserFun` from `Fun` to add constructors that instantiate the user-defined functoid implementation classes `UserImplA`, `UserImplB`, and so on, derived from `FunImpl`. The framework predefines several functoid implementation classes: `Global` implements wrappers around nonmember functions; `Compose` implements functoids resulting from composition; `Apply21`, `Apply31`, and so on, implement functoids resulting from partial application of functoids with more than one argument such that the resulting functoid takes the remaining single argument; finally, `Curry1` implements functoids resulting from "currying" functoids with more than one argument with respect to the first argument.

The sub-frameworks for two or more arguments have a similar structure. In the case of K arguments, there is an abstraction class `FunK` and an abstract implementation class `FunKImpl`. As in the case of a single argument, users derive from both framework classes. Again, there are various predefined functoid implementation classes: `GlobalK` implements wrappers around nonmember functions of K arguments. For $N > K$, `ApplyNK` implements functoids resulting from partial application of a functoid of N arguments to $N - K$ arguments. Since the resulting functoids take the remaining K arguments, the class `ApplyNK` is a subclass of `FunKImpl`. For $K \geq 2$ the class `CurryK` describes curried functoids that take K arguments one at a time. The actual currying is carried out by the corresponding member function `curryK` in the class `FunN`, where $N > K$. As function composition is defined only for functions of one argument, we do not consider it for $K \neq 1$.

We have not yet addressed the case of functions of zero arguments. We could treat functions without arguments as functions with one dummy argument of an enumerated type `unit` with a single value, but this approach would cause difficulties when creating wrappers for global functions with no arguments. We therefore provide a separate, simple sub-framework for this case consisting of only three class templates, `Fun0`, `Fun0Impl`, and `Global0`, whose roles are similar to the corresponding classes in the other cases. These class templates are parameterized only by the result type of the function.

We illustrate the structure of the framework in the notation used by Gamma et al. [GHJV93], which is an extension of the OMT (Object Modeling Technique) notation [R+91]. The framework for zero arguments is shown in Figure 1. The framework for one argument is shown in Figure 2. The framework for two or more arguments is shown in Figure 3, where K is the number of arguments. For simplicity, the classes `CurryK` are not shown.

A small example

Now that we have described the framework in detail, it is time to look at an example that illustrates the various features. Besides composition and partial application, the following example illustrates conversion from and to nonmember functions; we present the implementation of conversion to nonmember functions in Section 4. The functoid `Add` simply adds two numbers.

```
int add7(int x) { return x + 7; }

int callf(int(* f )(int)) { return f(9); }

class Add : public Fun2Impl<int,int,int>
{
public:
    virtual int operator()(int x, int y)
        const
    { return x + y; }
    virtual Fun2Impl<int,int,int>* copy()
        const
    { return new Add; }
};

main()
{
    // conversion from nonmember function
    const Fun<int, int> f(add7);

    const Fun2<int,int,int> g(new Add);

    cout << add7(3) << endl;
    cout << callf(add7) << endl;

    // f and add7 are now equivalent
    cout << f(3) << endl;
    // conversion to nonmember function
    cout << callf(f) << endl;

    // partial application
    cout << g(11)(3) << endl;
    // conversion to nonmember function
    cout << callf(g(11)) << endl;

    // composition and partial application
    cout << compose(f,g(11))(5) << endl;

    // currying and composition
    cout << compose(g.curry1(),f)(11)(5)
         << endl;
}
```

3 Case Study: The Same-Fringe Problem

The purpose of this section is threefold. First, it demonstrates how functional programming styles can be incorporated directly in C++ programs. Second, it serves as a case study that shows the practical usefulness of our system. Third, it disproves claims that this style of programming is not supported by C++ [Bak93].

The same-fringe problem

The *fringe* of a finite tree is the enumeration of its leaves in left-to-right order. The *same-fringe* problem is the problem of deciding whether two finite trees have the same fringe. In practice, this problem occurs when comparing for equality two trees that store data only in their leaves. A brute-force solution to this problem would involve generating the fringe of each tree as a list and then comparing the two lists for equality. This shortcoming of this solution is that it goes through considerable work to construct the entire lists although there might be a mismatch at the beginning of the lists. We could do slightly better by constructing the fringe of only one tree and iterating through the other tree.

A far better solution to the same-fringe problem is to compare the first leaf of each tree and continue only if they match. Such a solution could be expressed in terms of coroutines, which would need unbounded storage to keep track of the current path in the tree. These coroutines could be modeled by external iterators in C++. The drawback of this approach is that the tree traversal has to be made explicit instead of implicit and recursive.

A solution in a functional language

In a functional language, this problem could be solved elegantly in terms of *lazy streams* [FW76]. A lazy stream is a recursive data structure that is either an empty stream or a data item paired with a function that evaluates to another stream when invoked. This technique allows us to *delay* the generation of the entire fringe: we seemingly construct the fringe like an ordinary list, but the actual construction is performed on demand. In the functional language ML [MTH90], a data structure for lazy streams could be defined as follows. The two cases are called `Nil` and `Cons` in analogy to ordinary lists in functional languages. In ML, functions without arguments take a single argument of type `unit`.

```
datatype 'a Stream =
    Nil
  | Cons of 'a * (unit -> 'a Stream)
```

We first deal with the question of generating the fringe of a binary tree in form of a lazy stream. A binary tree is either a leaf containing an item or a node joining two subtrees together.

```
datatype 'a Tree =
    Leaf of 'a
  | Node of 'a Tree * 'a Tree
```

We now generate the fringe of a tree recursively. If the tree is a leaf, then the fringe is simply the pair of

the item and a function evaluating to an empty stream. Otherwise the tree is a node, and the fringe is the concatenation of the fringes of the subtrees. We actually concatenate two functions that evaluate to fringes when invoked to delay generating the entire fringes until requested. ML uses pattern matching to examine the structure of function arguments. The form "`fn () => expr`" is used to create an anonymous function closure on the fly. The comments identify the three different cases of anonymous functions we are creating.

```
fun fringe (Leaf x) =
    Cons(x, fn () => Nil)       (* Case 1 *)
  | fringe (Node(l,r)) =
    concat (fn () => fringe l)
           (fn () => fringe r) (* Case 2 *)
```

The concatenation of the two functions follows. If the first function evaluates to an empty stream, the fringe is simply the invocation of the second function. Otherwise the fringe is the first item of the first fringe paired with the concatenation of the rest of the first fringe and the second fringe:

```
fun concat f g =
    case f () of
      Nil         => g ()
    | Cons(x, h) =>              (* Case 3 *)
        Cons(x, fn () => concat h g)
```

The next job is to compare two lazy streams for equality. If both are empty, then they are equal. Otherwise, their first items have to match and the remaining streams have to be equal. In all other cases, the two streams are not equal. The following recursive function captures this notion of equality:

```
fun eq Nil            Nil            =
    true
  | eq (Cons(v1, f1)) (Cons(v2, f2)) =
    (v1 = v2) andalso eq (f1 ()) (f2 ())
  | eq s1             s2             =
    false
```

Now we are ready to define the samefringe function for two trees:

```
fun samefringe t1 t2 =
    eq (fringe t1) (fringe t2)
```

For example, among the following three trees, `t1` and `t2` have the same fringe, although they do not have the same shape, whereas `t0` has a different fringe:

```
val t0 = Node(Node(Leaf 3, Leaf 4),
              Node(Leaf 5, Leaf 7))
val t1 = Node(Node(Leaf 3, Leaf 4),
              Node(Leaf 5, Leaf 6))
val t2 = Node(Node(Node(Leaf 3, Leaf 4),
                   Leaf 5),
              Leaf 6)
```

Translating the solution to C++

We show how to translate the ML solution directly into C++ using the functoid framework. For the sake of simplicity, we deal only with integer items, but we could also have used templates for the various classes. Assume we are given a tree class with the following public member functions:

```
class Tree
{
public:
    int label() const;
    bool isleaf() const;
    const Tree& left() const;
    const Tree& right() const;
    ...
};
```

Our first task is to express lazy streams in C++. One approach to representing a recursive data structure in C++ is as a tagged union, using an enumerated tag field to indicate which of the cases an object belongs to and providing data members for all components of the data structure. We choose a better, more object-oriented approach that models each case of the data structure as a different subclass of a class for the data structure itself. To facilitate passing streams by value, we again employ the envelope/letter idiom. The class `Stream` becomes the envelope class, and we have an abstract letter class `StreamImpl` with concrete subclasses `NilStream` and `ConsStream` for the two cases of the data structure. We first present the class `Stream`.

```
class Stream
{
private:
    StreamImpl* theStream;
```

We need constructors for both cases, a copy constructor, and a destructor. The constructors take as arguments the components of the corresponding cases of the data structure. We assume a forward declaration of the class `Delay` for functions evaluating to streams.

```
public:
    Stream();
    Stream(int hd, const Delay& tl);
    Stream(const Stream& s)
        : theStream(s.theStream->copy()){ }
    ~Stream() { delete theStream; }
```

Now we need to design an interface for the stream class that allows us to distinguish between the two alternatives and to extract the components in the second case. The function `empty` tells us whether a stream is empty; in the nonempty case, `head` extracts the item, and `tail` extracts the function.

```
    bool empty() const
```

```
    { return theStream->empty(); }
    int head() const
    { return theStream->head(); }
    const Delay& tail() const
    { return theStream->tail(); }
    bool operator==(const Stream& s) const;
    Stream& operator=(const Stream& s);
};
```

Although we cannot actually implement the constructors, the destructors, and the equality operator until the class `Delay` is fully defined, we give their definitions at this point. The equality operator is a straightforward translation of the ML function `eq` given above.

```
Stream::Stream()
    : theStream(new NilStream) { }
Stream::Stream(int hd, const Delay& tl)
    : theStream(new ConsStream(hd, tl)) { }

bool Stream::operator==(const Stream& s)
    const
{
    if (empty() && s.empty())
        return true;
    else if (empty() || s.empty())
        return false;
    else
        return head() == s.head() &&
            tail()() == s.tail()();
}
```

Next, we present the abstract letter class `StreamImpl`. Its pure virtual member functions correspond to the member function of class `Stream`.

```
class StreamImpl
{
public:
    virtual ~StreamImpl() { }

    virtual bool empty() const = 0;
    virtual int head() const = 0;
    virtual const Delay& tail() const = 0;

    virtual StreamImpl* copy() const = 0;
};
```

We now define the two subclasses corresponding to the two cases of the data structure. These subclasses implement the pure virtual member functions defined in class `StreamImpl`. For brevity, we omit the `copy` member function, which simply duplicates the receiver. A `NilStream` is always empty and does not have a defined head or tail.

```
class NilStream : public StreamImpl
{
public:
    virtual bool empty() const
    { return true; }
    virtual int head() const { abort(); }
    virtual const Delay& tail() const
    { abort(); }
};
```

A `ConsStream` is never empty. The `head` and `tail` member functions return the corresponding data members, a number and a function, respectively.

```
class ConsStream : public StreamImpl
{
public:
    ConsStream(int x, const Delay& f)
        : hd(x), tl(f) { }
    virtual bool empty() const
    { return false; }
    virtual int head() const { return hd; }
    virtual const Delay& tail() const
    { return tl; }
private:
    int hd;
    Delay tl;
};
```

Now we must define the class `Delay`, which in turn depends on the stream class. We integrate this class in the functoid framework. The class `Delay` is a subclass of an appropriate instance of the class template `Fun0`, and the implementations of `Delay` will be subclasses of instances of the class template `Fun0Impl`. The purpose of introducing the class `Delay` is to capture the mutual dependency with the class `Stream` and to introduce appropriate constructors for each implementation of this class that we want to create. In the ML solution above we identified three cases of anonymous function closures that correspond to three implementations of the class `Delay`.

```
class Delay : public Fun0<Stream>
{
public:
    Delay();
    Delay(const Delay& f, const Delay& g);
    Delay(const Tree<int>& t);
};
```

We are going to implement the three cases as subclasses of `Fun0Impl`. We again omit the `copy` member function. Case 1 is a function of the form "`fn () => Nil`" evaluating to an empty stream. It is represented by the following functoid:

```
class EmptyDelay : public Fun0Impl<Stream>
{
public:
    virtual Stream operator()() const
    { return Stream(); }
};
```

Case 2 is a function of the form "`fn () => fringe t`" evaluating to the fringe of a tree. The corresponding functoid `FringeDelay` stores the tree `t` and invokes the function `fringe`,

a direct translation of the corresponding ML function.

```
Stream fringe(const Tree& t)
{
    if (t.isleaf())
        return Stream(t.label(), Delay());
    else
        return concat(Delay(t.left()),
                      Delay(t.right()));
}
class FringeDelay : public Fun0Impl<Stream>
{
public:
    FringeDelay(const Tree& t)
        : tree(t) { }
    virtual Stream operator()() const
    { return fringe(tree); }
private:
    const Tree& tree;
};
```

Case 3 is a function of the form "`fn () => concat g h`". The associated functoid `ConcatDelay` stores the two functions evaluating to the streams to be concatenated and invokes the function `concat`, again a translation of the corresponding ML function.

```
Stream concat(const Delay& f,
              const Delay& g)
{
    Stream s = f();
    if (s.empty()) return g();
    else return Stream(s.head(),
                       Delay(s.tail(), g));
}
class ConcatDelay : public Fun0Impl<Stream>
{
public:
    ConcatDelay(const Delay& f,
                const Delay& g)
        : fdelay(f), gdelay(g) { }
    virtual Stream operator()() const
    { return concat(fdelay, gdelay); }
private:
    Delay fdelay, gdelay;
};
```

Finally, we give the implementations of the three constructors for the class `Delay`. Each constructor creates an instance of the corresponding implementation class of the class `Delay`.

```
Delay::Delay()
    : Fun0<Stream>(new EmptyDelay)
{ }
Delay::Delay(const Delay& f,
             const Delay& g)
    : Fun0<Stream>(new CombineDelay(f,g))
{ }
Delay::Delay(const Tree<int>& t)
    : Fun<unit,Stream>(new FringeDelay(t))
{ }
```

We can now determine whether two trees have the same fringe by generating the corresponding streams and checking them for equality.

```
bool samefringe(const Tree& t1,
                const Tree& t2)
{ return fringe(t1) == fringe(t2); }
```

We extend the tree class in two ways. If we define equality of trees as having the same fringe, we can add the following equality operator to the class `Tree`:

```
bool Tree::operator==(const Tree& t) const
{ return fringe(*this) == fringe(t); }
```

Furthermore, we can enhance the class `Tree` with an external (active) iterator class that traverses the fringe of the tree. This class `TreeIterator` enables us to define more than one iterator on the same tree.

```
class TreeIterator
{
public:
    TreeIterator(const Tree& t)
        : theTree(t) { restart(); }
    operator bool() const
    { return ! theFringe.empty(); }
    int current() const
    { return theFringe.head(); }
    void next()
    {
        assert(! theFringe.empty());
        theFringe = theFringe.tail()();
    }
    void restart()
    { theFringe = fringe(theTree); }
private:
    const Tree& theTree;
    Stream theFringe;
};
```

The `next` function uses the assignment operator for the class `Stream`, which we define using the `copy` function.

```
Stream& Stream::operator=(const Stream& s)
{
    if (this != &s)
    {
        delete theStream;
        theStream = s.theStream->copy();
    }
    return *this;
}
```

4 Converting Functoids to Ordinary Functions

The framework presented in Section 2 falls short of the requirement that functoids be convertible to ordinary nonmember functions. This shortcoming

stems from a fundamental difference between member functions and nonmember functions, which precludes us from simply using a pointer to the function call operator of a functoid as an ordinary function.

The heterogeneity problem

The fundamental difference between member functions and nonmember functions was recognized by Young [You92] and is called the *heterogeneity problem* by Dami [Dam94]. Technically, a call to a nonmember function requires a stack pointer to store the actual arguments and the address of the function to be called. A member function invocation, on the other hand, requires a stack pointer to store the arguments, the address of the member function, and the address of the receiver. This fundamental difference in the calling mechanism makes it impossible to use a member function where a nonmember function is expected, for example, as a callback from an existing class library.

The proposed solutions [Fek91, You92, CL95] require that the programmer writes a nonmember function that explicitly invokes the C++ member function from a specific receiver. This solution is generally not very good because the programmer has to write a wrapper for every combination of a member function and a receiver to be used as a callback. More seriously, this solution does not work at all for the framework because we create functoids on the fly and thus cannot anticipate what wrappers to provide.

The solution using partial binding

Rescue comes in the form of a solution proposed and implemented by Dami [Dam94], which addresses a more general partial binding problem. In this solution, when we perform a partial binding, we create a data structure that stores the address of the function, the arguments, and code to complete the bindings and invoke the function later. This approach is compiler- and machine-dependent; it currently works with the GNU CC compiler [Sta94] on NeXT and Sparc architectures, but could be ported to other languages, compilers, or architectures. A similar mechanism that maps Scheme closure objects to C functions is described by Rose and Muller [RM92]. The ObjectKit system for ParcPlace Smalltalk allows passing Smalltalk objects, including closures, to C functions [RM92, quoting P. Deutsch].

From the programmer's perspective, Dami's mechanism consists of the function `curry`, whose arguments are a pointer to the memory where the data structure should be allocated, the function to be invoked, the total number of arguments, and the number of arguments supplied here, and those arguments.

```
typedef void*(* anyFunc )();

extern anyFunc curry(void* mem, anyFunc f,
                     int nargs, int cargs, ...);
```

This mechanism extends to object-oriented languages in the sense that the receiver of a message is an (implicit) first argument to the method invoked. We can thus convert a member function to a nonmember function by partial application to the receiver `this`. While the mechanism itself is not type-safe, we safely hide it inside the framework, and the user only sees it as a type conversion operator of functoids back to nonmember functions. We describe how the mechanism is implemented for functoids with a single argument; the implementation for functoids with more arguments is analogous. The class template `FunImpl` gets an additional member function that performs the conversion of its function call operator to a nonmember function.

```
typedef Out(* FunPtr )(In);

virtual FunPtr FunImpl<In,Out>::cfun()
const
{
    return FunPtr(curry(0,
        anyFunc(this->operator()),
                3, 1, this));
}
```

The class template `Fun` is extended by a type conversion operator that invokes `cfun` on the implementation of the functoid. To make sure that the conversion is executed only once, we store the resulting function pointer in an additional data member `fun` of `Fun`, which the constructors initialize to the null pointer. The associated data must be deallocated using the `free` function when the functoid is destructed. The new members of `Fun` are as follows.

```
template <class In, class Out> class Fun
{
public:
    typedef Out(* FunPtr )(In);
    ...
    ~Fun()
    { delete impl; if (fun) free(fun); }
    operator FunPtr() const
    {
        if (fun == NULL)
            ((Fun<In,Out>*) this)->fun =
                impl->cfun(); return fun;
    }
    ...
```

```
private:
    FunPtr fun;
};
```

Now our conversion requirement is satisfied both ways, and functoids and nonmember functions are indistinguishable to the user. The example at the end of Section 2 illustrates this feature.

5 Conclusion

We have presented a type-safe generalized framework that supports higher-order functional programming styles within C++ programs. The framework is implemented entirely in the form of C++ class templates, except for a compiler- and machine-dependent mechanism for converting member functions to nonmember functions [Dam94]. The framework could be translated to other object-oriented languages that support both inheritance and genericity.

The main issues in the assessment of our framework are expressiveness and efficiency. To address the first issue, we compare our framework to existing functional programming languages.

It is a fundamental limitation of most class-based object-oriented languages that each distinct behavior must be given a class name [Ros95a]. Consequently, our framework does not provide a mechanism for creating anonymous function closures on the fly. This is in contrast to functional languages, in which anonymous closures are routinely passed to and returned from functions. Rose [Ros95b] describes an extension of C++ with parameterless anonymous functions called *thunks*; a thunk can be converted to a parameterized function by specifying which variables used in the body of the thunk are to be treated as parameters.

Another limitation of the functoid idiom in general, not just of the functoid framework, is that the programmer must establish and maintain an explicit correspondence between variables used in the body of the closure and instance variables of the functoid. By contrast, functional and other languages with block structure and nested functions, such as Algol or Pascal, automatically capture all local variables that are used in the closure. Breuel [Bre88] solves this shortcoming in C and C++ by allowing functions to be nested. Thunks [Ros95b] provide a solution as well.

Another drawback of the functoid framework stems from the way type information is required in instantiations of C++ class templates. While the examples presented in this paper do not require lengthy type parameters, the type information required in more complex applications of the framework is likely to get out of hand, especially when higher numbers of arguments are involved. Dami [Dam95] suggests extending the compiler to keep track of the required type parameters automatically.

There are several sources of inefficiency in the framework as compared to typical implementations of functional languages. First, we use call-by-value to facilitate memory management. This approach requires a considerable amount of copying, depending on the size of the functoid implementations involved. The problem could be addressed by improving the memory management strategy, for example, by using garbage collection for functoids. Memory management could still be hidden from the user by overloading the `new` and `delete` operators for functoids. Second, the structure of the framework requires a virtual function call operator that is overridden in the user classes to allow dynamic selection of the appropriate functoid implementation. This problem is inherent in the design of the framework and has no simple solution. Third, unlike in functional languages, function closures are controlled by the programmer instead of the compiler. This precludes the sort of optimizations a compiler of a functional language would apply.

Other approaches that combine functional languages and C++ include an interpreter accessible within C++ [Kla93] and an interpreter written in C++ [RK88]. A detailed comparison with our work would go beyond the scope of this paper. While the translation outlined informally in Section 3 is not suitable at present as an efficient implementation of functional languages, the paper demonstrates that the framework provides access to various functional idioms within object-oriented languages.

Acknowledgments

Special thanks go to Gerald Baumgartner for his careful review of the initial version of this paper, to Laurent Dami for helpful comments and for providing his Curry code, to Thomas Kühne for stimulating email discussions on this and related material, and to John Rose for valuable comments and for sharing his insights on thunks with me.

References

[App89] Apple Computer, Inc., Cupertino, CA. *Macintosh Programmers Workshop Pascal 3.0 Reference*, 1989.

[Bak93] H. Baker. Iterators: Signs of weakness in

object-oriented languages. *ACM OOPS Messenger*, 4(3):18–25, July 1993.

[Bor94] Borland, Inc. *Borland C/C++ 4.0 Reference Manual*, 1994.

[Bre88] T. Breuel. Lexical closures for C++. In *Proc. USENIX C++ Conf.*, pages 293–304, Denver, CO, October 1988.

[CL95] M. Cline and G. Lomow. *C++ FAQs*. Addison-Wesley, 1995.

[Cop92] J. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[Dam94] L. Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Université de Genève, April 1994.

[Dam95] L. Dami. Adding closure support to the C++ compiler. Personal communication, March 1995.

[ES90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Fek91] J. Fekete. WWL, a widget wrapper library for C++, 1991. Laboratoire de Recherche en Informatique, Orsay.

[FW76] D. Friedman and D. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming*, pages 257–284. Edinburgh University Press, 1976.

[GHJV93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1993.

[HS87] S. Harbison and G. Steele. *C: A Reference Manual*. Prentice-Hall, 2nd edition, 1987.

[JF88] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[Kla93] H. Klagges. A functional language interpreter integrated into the C++ language system. Master's thesis, Balliol College, University of Oxford, University Computing Laboratory, September 1993.

[Küh95] T. Kühne. Inheritance versus parameterization. In Christine Mingins and Bertrand Meyer, editors, *Proc. Technology of Object-Oriented Languages and Systems (TOOLS Pacific '94)*, pages 235–245, Prentice Hall International, Inc., London, 1995. Prentice-Hall. For correct version ask author; proceedings contain corrupted version.

[LCI+92] M. Linton, P. Calder, J. Interrante, S. Tang, and J. Vlissides. *InterViews 3.1 Reference Manual*. CSL, Stanford University, 1992.

[Mey92] S. Meyers. *Effective C++*. Addison-Wesley, 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Red95] U. Reddy. The design of Core C++. Unpublished draft, March 1995.

[RK88] V. Russo and S. Kaplan. A C++ interpreter for Scheme. In *Proc. USENIX C++ Conf.*, pages 95–108, Denver, CO, October 1988.

[RM92] J. Rose and H. Muller. Integrating the Scheme and C languages. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 247–259, San Francisco, 1992.

[Ros95a] J. Rose. C closures. Personal communication, April 1995.

[Ros95b] J. Rose. Functional programming and call-by-name in C++. Personal communication, April 1995.

[R+91] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Sta94] R. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, Massachusetts, September 1994. Available as part of the GCC-2.6.3 distribution.

[VL90] J. Vlissides and M. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++ — An object-oriented application framework in C++. In *Proc. ACM Conf. Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, pages 46-57, San Diego, CA, 1988.

[You92] D. Young. *Object-Oriented Programming with C++ and OSF/Motif*. Prentice-Hall, 1992.

Lingua Franca: An IDL for Structural Subtyping Distributed Object Systems

Patrick A. Muckelbauer and Vincent F. Russo
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{muckel|russo}@cs.purdue.edu

Abstract

Recently the trend has been towards applying object-oriented techniques to address problems of building scalable and maintainable distributed systems. Object-oriented programming increases modularity and data abstraction by supporting encapsulation through narrow, rigidly defined and strongly enforced interfaces to objects. Unfortunately, object-oriented interfaces and mechanisms are usually only accessible and enforced through language mechanisms or strict programming conventions. This severely limits the degree to which disjoint, unrelated components can interact in a multilingual, loosely coupled distributed system. An accepted solution to the language dependence problem is the use of high-level interface description languages (IDLs). IDLs provide a description mechanism for an object's interface that is independent of any programming language. In this paper we describe an interface description language and runtime support system that uses structural subtyping rules rather than the traditional interface name equivalence rules for conformance checking. We argue that the choice of structural subtyping rather than interface name equivalence leads to a less coupled and more extensible distributed system.

1 Introduction

As part of the *Renaissance* project at Purdue University we are researching tools and techniques for building scalable, maintainable and extensible distributed systems. The difficulty of building such systems is due mainly to the high degree of heterogeneity in a distributed environment. Large scale distributed systems must contend with multiple architectural platforms and the differences in their data representations, multiple languages and the differences in their notion of types and values, and multiple vendors providing different implementations and versions of components.

In order to build a successful and scalable distributed system in such an environment we believe it is necessary to minimize the *coupling* between distributed components[4]. Coupling measures the interdependencies between interacting components. Low component coupling is a desirable feature of any system because it decreases the difficulty of separating, understanding, maintaining and reusing individual components[10]. Over time, it is inevitable that some components of a system will need to evolve. High component coupling increase the likelyhood that, in order to maintain inter-operability, changes to support this evolution will require modifications to other dependent components. These modifications may, in turn, require modifications to other dependent components and so on. In highly distributed systems this problem will be worsened since numerous programmers, often working for different vendors, will develop individual components. Furthermore, these programmers are often unaware of many component inter-dependencies.

Recent trends have been towards applying object-oriented techniques to address such problems in distributed and non-distributed systems. Object-oriented programming increases program modularity and data abstraction by supporting encapsulation through narrow, rigidly defined and strongly enforced interfaces to *objects*[4]. An object consist of an encapsulated state and a set of operations, or *methods*, to modify or access the state. The *interface* of an object is the collection of all its methods. The *only* way to modify or access an object's state is to *invoke* a method of the object's interface. We term an autonomous collection of objects a *domain*. The increased abstraction and encapsulation provided by the object-oriented model helps reduce component

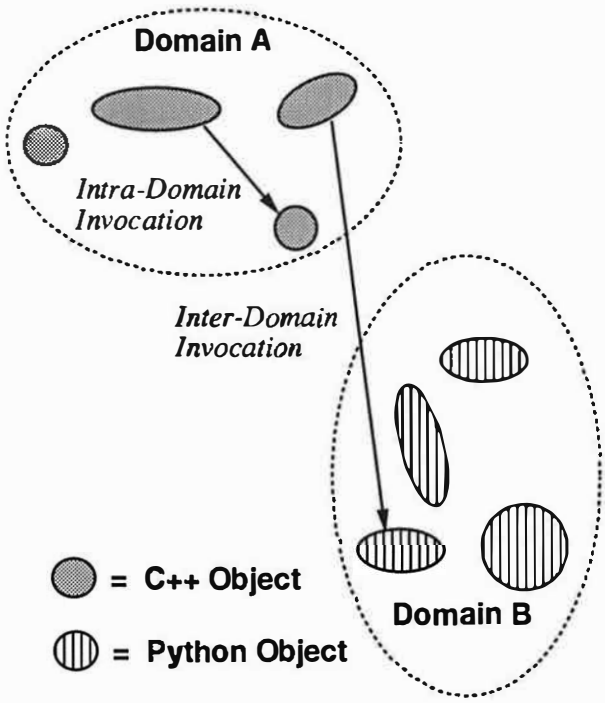


Figure 1: Inter/Intra-domain Method Invocations

coupling across separately developed domains.

Like other recent work[19], we use a distributed object system model in which interaction between clients and servers are implemented as method invocations on objects in separate domains. I.e., clients access a server by invoking methods on remote references to objects in the server's domain. Each domain is self-contained and usually implemented in a single programming language (or at least a single consistent object model). Intra-domain object method invocations are supported by the language(s) in which the domain is implemented. Our work is to develop tools and libraries that provide support for inter-domain object method invocations. This interaction is illustrated in Figure 1.

The modularity and encapsulation provided by the object-oriented model would at first glance seem to be ideal for distributed systems. Unfortunately, object-oriented mechanisms are usually only accessible and enforced through language support or strict programming conventions. This severely limits the degree to which disjoint, unrelated components can interact in a multilingual, loosely coupled distributed system.

An accepted solution to the language dependence problem is the use of high-level interface description languages (IDLs)[19]. IDLs provide a mechanism for specifying an object's interface independently of any programming language. Translators map these specifications into a target language's notion of objects and interfaces. The generated language specific modules are used by programmers to implement server objects (usually by inheriting from automatically generated parent classes) and/or to generate typed references to remote objects.

Components in a distributed system that wish to interact need to agree on common interface descriptions in order to guarantee the proper use of objects. The checking of this agreement, which is termed *conformance* checking, defines rules for the substitutability of objects, i.e., when a server object provides the interface that a client domain is expecting. A common approach to checking conformance is to define a server object's *type* as a named interface that it exports. Client requirements, also specified as named interfaces, are compared to server objects' types during conformance checking. I.e., conformance is defined to to be equivalence between interface type names (or hierarchies of interface type names). Consequently, interacting domains must agree a priori upon the names of the interfaces and their definitions.

As the conformance rules for our system, we chose instead to use structural subtyping rules. In this paper we will argue that the choice of structural subtyping over name equivalence leads to less coupled and more extensible systems.

The remainder of the paper will first cover some background material and then introduce our system by discussing the *Lingua Franca* Interface Description Language and the implementation of the runtime support libraries. We then discuss the differences between name based conformance checking and structural subtyping and detail the conformance rules we have chosen. We follow with discussions of the performance of method invocation and runtime conformance checking. Finally, we end with a comparison to related work and a summary of the advantages and disadvantages of structural subtyping.

2 Background

A distributed system is an environment that allows programs to interact across address space and machine boundaries. This is accomplished by supporting access to values within a remote domain through *remote references* to that domain. Operations on remote reference allow accessing and modifying remote values. A program providing remote values is termed a *server*. A program accessing those values

is termed a *client* of the server.

Most distributed systems introduce some notion of *types* for describing the server values accessible through remote references. Types provide structure to a system by placing constraints on the set of values servers can export and clients can reference. Checking conformance to these constraints guarantees that remote references are used in a consistent manner and amounts to testing whether the type of a server value conforms to the type of a client's remote references to the value.

The inter-operability between a client and server is determined by the conformance rules chosen for the system. The conformance rules provide a substitution rule for types. If a type t_x conforms to a type t_y then a value of type t_x exported by a server can be referenced in a type consistent manner by a client expecting a value of type t_y. Specifically, t_x is said to be a *subtype* of t_y and t_y is said to be a *supertype* of t_x.

Conformance checking in a distributed system has several advantages: it catches errors early, it guarantees a sense of correctness to the system by not allowing clients and servers to interact in an inconsistent manner, and it allows greater execution time efficiency of remote accesses to be achieved[6].

Client programs bind to server values dynamically at runtime. Since the types of these server values are not known at compilation time, a static analysis of the client domain is not sufficient to check the conformance between the types expected by the client's remote references and the types of the remotely referenced values. Instead, distributed systems must rely on runtime checks to test conformance and guarantee type consistency. Before a remote reference can be used, the referenced value's conformance to a type specified within the client must be checked. We term this operation *narrowing*. Once a reference has been narrowed, static analysis of the client's code can further guarantee type consistency. The narrow procedure takes a remote reference and a local representation of a type. If, using a set of chosen rules, the type of the remote value conforms to the locally specified type, a new typed remote reference to the value is generated. The server value can then be accessed in a type-safe manner through this new typed reference with no further runtime checks. The narrowing operation usually returns a NULL reference, or perhaps raises an exception, to indicate the conformance check failed. There are several important consequences of the narrow procedure. First, there must be some runtime representation for types, secondly, a client must be able to retrieve or query a remote value for its runtime type information, and thirdly, a sound set of conformance rules must be defined. Conformance rules are sound if the rules do not lead to type inconsistences.

3 System Overview

A detailed specification of our rules for conformance are discussed in Section 4.3 but, briefly, our notion of an object's "type" is simply its interface specified as a set of named methods and their argument and result types. Client requirements, specified as interfaces *expected*, are compared at runtime to exported server interfaces using the standard subtype notion of method subset matching with covariance of return types and contravariance of arguments. I.e., interface I_x *conforms* to interface I_y if and only if every method in I_y is also found in I_x (I_x may have additional methods as well), and for every method in I_y, each of the following conditions hold.

1. The corresponding method in I_x has the same number of parameters.
2. The parameters of the method in I_y conform to the corresponding parameters in the method in I_x.
3. The result of the method in I_x conforms to the result of the corresponding method in I_y.

The names given to the interfaces (but not the individual methods in the interface) are irrelevant to the conformance check and are local only to a particular application.

The major tools we have implemented to support our system model are the *Lingua Franca* Interface Description Language and a set of runtime libraries to support inter-domain method invocations between domains described with *Lingua Franca* programs. Using our tools, a server object implementor describes the object interfaces exported by his/her domain in *Lingua Franca* and uses a translator to convert these descriptions into the target programming language in which the domain will be implemented. *Separately*, client programmers describe their requirements with *Lingua Franca* descriptions specifying the interfaces their applications need from server domain objects. Our approach differs from others in that the two descriptions are separate and can be independently modified, extended, versioned and even named. The conformance between a client and server interface is determined at runtime during narrowing when runtime representations of these two interfaces are structurally compared. The runtime support libraries implement the necessary

runtime type information, the runtime conformance checking (narrowing), and object invocations across distributed modules. The libraries make no assumptions about the underlying hardware. They define a canonical representation for values and provides mappings to and from the canonical representation and the target hardware representations.

It is obvious that the approaches we use to obtain hardware, language, and implementation independence are very similar to the approaches used in other distributed systems, and we have leveraged heavily from their experiences. The main focus of our work is how to support *interface independence* and *versioning* in a way that minimizes component (client and server) coupling. Interface independence (low interface coupling) allows the minimization of the effects changes to a server's interface have on the clients of the server. Interface independence is achieved through the subtyping relations defined by the chosen conformance rules. The more flexible the conformance rules, the more tolerant the system is to changes to an object's interface. Versioning is the ability to join interfaces (versions) while still conforming to clients using the individual interfaces. I.e., it should be possible to allow a single server to provide multiple versions of a service with possibly different semantics. The extent to which both of these can be achieved depends on the conformance rules used and mechanism for specifying interfaces.

3.1 Lingua Franca

The *Lingua Franca* type description language provides a simple but powerful mechanism for describing types in a distributed environment. By keeping *Lingua Franca* simple we hope to reduce the burden of learning a new language. Simplicity is achieved by starting with a small number of basic types and using a set of powerful *type constructors* to build more complicated types. The design of *Lingua Franca* attempts to adhere to the principal of *Data Type Completeness*: wherever a type is allowed in a type constructor, any type may be used without exception, including other (inline) type constructors. This allows a rich set of types to be described using only a few constructors and reduces the number of rules and the overall complexity of the language.

Lingua Franca supports both *interface types* that describe object interfaces and *data types* that describe non-methoded values. *Lingua Franca* also allows an *Anything* type that is the infinite union of all types (i.e. the top of the type lattice). Type *Anything* allows all types, whether interface or data, to be treated uniformly.

Types are created from a small set of primitive types and a set of type constructors for describing more complex types such as sequences, records, and, most importantly, object interfaces. Translators map *Lingua Franca* programs and the types they describe into specific target languages producing language specific types. For interface types, the language specific types (usually a class) are used by programmers to construct servers (via subclassing of automatically generated parent classes) and/or to generate typed references to remote objects after narrowing.

Lingua Franca separates issues and details of how *Lingua Franca* types are represented in a target language from the specification of those types. A set of default mappings to implementation types is built into each translator. For example, sequences are translated into arrays in C++[23]. Programmers can also provide directives to the translator to control the representation of *Lingua Franca* types within a target language. These directives are of the form:

Implement S **as** T

where S is a *Lingua Franca* type and T is a language specific type. Supporting customizable translation provides for a clean separation of type and representation and leads to a simplier language with greater flexibility. For instance, *Lingua Franca* does not support separate type constructors for lists and arrays, but instead, only supports a type constructor for sequences of a type. Whether a sequence is represented as a linked list or an array in the target language is unimportant at the type *specification* level (i.e. within *Lingua Franca*) but can be selected with the **Implement** operation. The representation type is not part of the conformance check and can be different in a client and server.

Lingua Franca defines a single set of semantics for passing arguments and results during method invocation. Interface types (objects) are passed by reference. Data types are passed by value. Passing methoded objects by value is really an object mobility issue and involves moving the implementation and state of the object (and its closure) between programs. This information is currently outside the scope of *Lingua Franca*. Data types are passed by value so that they may be accessed by the remote domain efficiently. While it might be useful to allow programmers the choice of alternate semantics such as pass by reference or copy over/copy back, in practice this has not been necessary and the added flexibility does not seem to warrant the complexity it would add to the language's syntax and semantics, conformance rules, and runtime support libraries.

3.1.1 Grammar

A program written in *Lingua Franca* consist of statements for binding values to identifiers. Subsequent uses of an identifier in the program refer to its value and are permissible anywhere its value is. Two kinds of values within a *Lingua Franca* program are allowed: integer and type. An integer is represented as an arithmetic expression in infix notation with the standard rules of precedence. A type is created from a set of primitive types and the set of type constructors for creating more complex types. The syntax for a *Lingua Franca* program is

program ::= [<*statement*>] *

statement ::= <*type-stmt*>
| <*integer-stmt*>
| <*recursive-stmt*>

type-stmt ::= type <*ident*> = <*type*>
[, <*ident*> = <*type*>] * ;

integer-stmt ::= integer <*ident*> = <*expr*>
[, <*ident*> = <*expr*>] * ;

The *type-stmt* declares a set of identifiers, refered to as *type identifiers*, and binds a type value to each of them. Similarly the *integer-stmt* declares a set of identifiers, refered to as *integer identifiers*, and binds an integer value to each of them. The *recursive-stmt* is for building recursive types and is discussed in Section 3.1.3.

Identifiers share the same namespace and may not be redefined. Consequently, an identifier always refers to a same value. In this respect, identifiers should be thought of as a naming mechanism for values and not as a variable with a value. Finally, an identifier must be defined before it is used (the only exception to this rule is within a recursive statement).

Figure 2 lists the entire set of primitive types in *Lingua Franca* and the set of values they define. Records are built using the *record of* type constructor. Discriminated unions are built using the *case of* type constructor and are very similar in form to *datatypes* in ML[16]. A tag or discriminate field for a case type is not necessary as it is implicit in the declaration. Field names within the *case of* and *record of* type constructors must be unique.

Monomorphic sequences are built using the *sequence of* type constructor. Both fixed and variable length sequences can be built by optionally supplying a length to the *sequence of* type constructor.

Type	Description
Integer	a 4 byte scalar value
Byte	1 octet of raw data
Character	an ASCII character
Nil	the single element Nil
String	sequence of ASCII characters
Boolean	TRUE or FALSE
Float	IEEE floating point numbers

Figure 2: Primitive Types in *Lingua Franca*

An object interface is described using the *interface of* type constructor. All methods names of an interface must be unique. Several alternatives were considered for method overloading. Those based on the structure of the types of the arguments/results led to complex rules that did not warrant the small gain in programmer usefulness. Finally, the names of method arguments are optional and for documentation only. As this implies, the names of arguments are not used in conformance checking.

Lingua Franca does not have a separate type constructor for enumerated types, but rather, an enumerated type is treated as a special case of a discriminated union. An enumerated type is described with the *case of* type constructor where the type of the data associated which each tag name is `Nil`. For example the the enumerated type (*red*, *green*, *blue*) is specified as

```
case of
     red : Nil;
     green : Nil;
     blue : Nil;
end case
```

The complete syntax for a type specification is shown in Figure 3.

3.1.2 An Example

The small code fragment detailed below is a sample *Lingua Franca* program that describes a set of interfaces necessary to implement a simple file system. The program defines two interfaces: `FileServer` and `File`. The `FileServer` interface provides the typical operation found on a directory. In particular, opening, creating, removing, and aliasing files. The `File` interface provides methods for reading and writing files.

```
type ::=                 <type-identifier>        | <primitive-type>
                         | <infinite-union-type>  | <sequence-of-type>
                         | <pointer-to-type>      | <case-of-type>
                         | <record-of-type>       | <interface-of-type>
primitive-type ::=       Integer | Byte | Character | Nil | Float | String | Boolean
infinite-union-type ::=  Anything
sequence-of-type ::=     sequence of [ <expr> ] <type>
pointer-to-type ::=      pointer to <type>
case-of-type ::=         case of
                             [ <name> : <type> ; ] *
                         end case
record-of-type ::=       record of
                             [ <name> : <type> ; ] *
                         end record
interface-of-type ::=    interface of
                             [ <name> ( <argument-list> ) : <type> ; ] *
                         end interface
argument-list ::=        [ <argument> [ , <argument> ] * ]
argument ::=             [ <name> : ] <type>
```

Figure 3: Syntax for Type Specifications

```
type Buffer = sequence of Byte;
type File = interface of
   read( pos : Integer, len : Integer )
         : Buffer;
   write( Buffer, pos : Integer )
         : Integer;
end interface;
type FileServer = interface of
   open( name : String )
         : pointer to File;
   create( name : String )
         : pointer to File;
   remove( name : String )
         : Boolean;
   link( name : String, File )
         : Boolean;
end interface;
```

3.1.3 Recursive Types

Intuitively, a recursive type in one that is constructed by referencing itself. Let S be the set of types used to construct T. The closure of T is S unioned with the closure of each element of S. By definition, the closure of a primitive type is the empty set. A type T is defined to be recursive if it is in its own closure. In *Lingua Franca*, type identifiers must be defined before they are used. Consequently, recursive types can not be described without additional rules allowing, at least, the ability to refer to a type in its own construction.

A possible solution is to allow a type identifier to remain unbound during its own definition. This allows types that are self recursive to be described but cannot describe mutually recursive types. The types t_1 and t_2 are mutually recursive if t_1 is used in the construction of t_2 and conversely t_2 is used in the construction of t_1. What is needed is the ability to refer to unbound types or, in terms of *Lingua Franca*, to use an identifier before defining it. This is the purpose of the recursive statement. The recursive statement declares a set of type identifiers and, within this statement, allows them to be used before they are defined. The recursive statement has the syntax:

recursive-stmt ::= recursive <*type-stmt*>

Care must be taken so that the value described by a type has a finite representation. For example, if no restrictions are placed on the use of undefined identifiers within a recursive statement the following type, which describes values with no finite representation, could be specified:

```
recursive type
  Infinite = record of
                field1 : Integer;
                field2 : Infinite;
             end record;
```

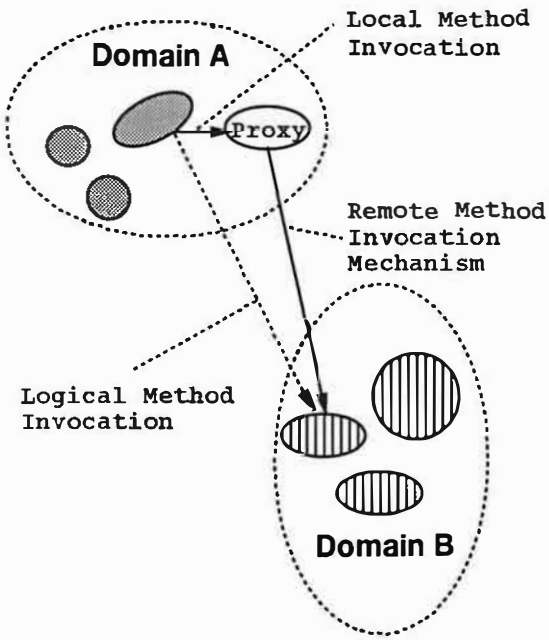


Figure 4: Proxy Object

To prevent this problem from occuring, the use of an undeclared type identifier within a recursive statement is limited to:

1. the type used in a pointer type constructor
2. an argument to or result of a method in an interface type constructor

3.2 The Runtime Support Libraries

In our system, all interaction between client and server domains are through method invocations on remote objects. All objects have a globally unique identifier. These identifiers can be thought of as *handles* to the objects. Passing an object between domains amounts to passing the object's handle. Transparent use of these handles is achieved through the use of *proxy* objects[22]. As shown in Figure 4, a proxy is a local representation of a remote object and maps the language notion of procedure call or method invocation transparently into a remote method invocation. In our system, the local proxy object is an instance of a class generated automatically from a *Lingua Franca* interface description. The code implementing the proxy, including the necessary argument marshalling and unmarshalling, is also automatically generated from this description.

The runtime system supports a universal operation applicable to all values described by *Lingua Franca* programs: the `signature` operation. This operation is used to retrieve the type information needed during narrowing. The `signature` operation returns a *signature object* that describes the type of the value to which it was applied (i.e. all values are self-describing). Signature objects and the data they contain are automatically generated from a *Lingua Franca* program by the language translators. The signature object provides a complete description of the type. For an interface type this includes the names, return types, and argument types of all its methods. For a record, or union type, this includes the names and types of each field. The signature object describing the type of a value provides sufficient information to create at runtime a new *Lingua Franca* program describing the type of the value. The new *Lingua Franca* program may differ syntactically from the original, but, the value's type in the new and original program will be structurally equivalent. The signature objects provide the necessary runtime type information to perform conformance checking. As an optimization, results of a conformance check are cached using the handles of the signature objects compared by the check.

The *Renaissance* system provides support for *Lingua Franca* derived programs running on UNIX systems or on a native object-oriented operating system kernel. Translators are currently supported to map *Lingua Franca* programs into C++[23] for compiled applications and Python[25] for interpreted applications. Additional translators of *Lingua Franca* to other programming languages are being investigated along with the corresponding runtime libraries. The *Renaissance* system and tools provide support for inter-domain object method invocations (transparently) using *Lingua Franca* object interface descriptions. To promote interoperability and scalability, no assumptions are made about what language is used within a domain or on what operating system or architecture the domain in running. For example, the current prototype supports UNIX application domains invoking methods of objects that are part of an object-oriented operating system kernel running on a separate machine (and vice-versa). We implemented this operating system as a proof of concept. The operating system's services (kernel, file system, network, devices, etc.) are all *Renaissance* domains.

The transport mechanisms used to perform cross-domain messaging are implemented as part of the *Renaissance* runtime support libraries and are modular and based upon the domains of the client and server[17]. Communication protocols have been designed and built to support machine independent,

cross domain method invocations using shared memory and UDP/IP packet transport. In cases where client and server turn out to be in the same domain, the object invocations are optimized to be local language method invocations, i.e., there are no proxies for properly conforming objects in the same address space.

4 Conformance Rules

We explore two approaches to conformance in a distributed system to see how they effect interface independence and versioning. The first approach, used by conventional large-scale distributed systems, we call *inheritance subtyping* and bases conformance on a hierarchy of type names. The second approach, used in our distributed system, is called *structural subtyping* and bases conformance on the structural compatability of the server's type and the client's requirements. We will show the choice of structural subtyping over inheritance subtyping to be more flexible.

For simplicity, all examples that follow use the *Lingua Franca* IDL described earlier. For illustrations of interface inheritance it is necessary to introduce additional syntax. The '+' type constructor will be used to denote interface inheritance and will only be used in examples using inheritance subtyping. Also, for the purposes of clarity in certain examples underscores in names will be used which, strictly speaking, *Lingua Franca* does not allow.

4.1 Inheritance Subtyping

A common approach to checking conformance is for each object to describe its interface with a name and to have agreement between interacting components as to what the names "mean". A mechanism is usually provided for interfaces to *inherit* from other interfaces. Interfaces that inherit from other interfaces are termed *subclasses* (for historical reasons) of those interfaces and this relationship applies transitively. A subclass' interface is a copy of the direct parent's interfaces plus any additional methods the subclass chooses to add. For inheritance subtyping, conformance is defined by the inheritance relationship between interfaces. Interface I_{sub} conforms to interface I_{super} if and only if either I_{sub} is I_{super} or I_{sub} is transitively a subclass of I_{super}. It is interesting to note that inheritance subtyping is really an extension of name *equivalence* that allows types to have a finite set of names.

Changes to the system are described through new named interfaces that are added to the mutually agreed upon hierarchy. As long as the new type of a server inherits from its old type, the clients of the server need not be modified. Interface independence is achieved only if the inheritance relationship described above exists between the interfaces.

Versioning of a server is easily handled with interface inheritance. Each version of the server is described by a separate named interface, and a server multiply inherits the versions of the service it is willing to provide. During narrowing, the client supplies the name of the version needed and if the server has inherited this version it conforms.

A potential problem with this solution is that several multiply inherited versions may have methods with the same name but with different semantics or different arguments. Fortunately, there exist rules for method overloading that allow for such conflicts. Method names can be disambiguated on the names of the types of arguments and/or the client can specify the name of the interface to which the method belongs.

4.2 Structural Subtyping

Rather than using an interface hierarchy for conformance, structural subtyping considers the type of an object as simply its interface specified as a set of named methods and their argument and result types. Client requirements, specified as a set of methods *expected*, are compared at runtime to server object interfaces using the standard subtype notion of method subset matching with covariance of return types and contravariance of arguments. The complete rules we have chosen for interface conformance are discussed in Section 4.3, but it is important to note that a client can chose to over specify the requirements of the arguments (it provides more than is expected by the server), and may under specify the requirements of results (it accepts back less than the server specifies).

Interface independence is achieved as a consequence of the subtyping rules used. Servers' interfaces may change and, as long as the changes do not effect the clients' compatability with these servers, the clients need not change. To reduce the dependency between clients and servers, clients should describe their requirements with the narrowest interface possible. This means specifying only the methods needed, overspecifying the arguments to these methods, and specifying their result types with the minimum requirements possible. This is not as easy to achieve with inheritance based subtyping.

At first glance versioning seems straight forward; an object can support multiple versions by describ-

ing it's interface as the union of the versions it wishes to provide. But again, there is the problem with method name clashes between versions. In the case of structural subtyping, it is not possible to rely on the names of interfaces or the names of the types of the arguments to aid in the disambiguation. Our solution is to consider an object's type not as a single interface but rather a set interfaces (i.e. a set of versions). During narrowing, each of the server object's interfaces is tested for conformance to the client's interface. If at least one of the interfaces conforms, the server object conforms to the client requirements. The interfaces can be sorted by version number from most recent to least recent. If the multiple versions conform to the client's interface the most recent one is chosen. Note, the type of the object is not the union of the interfaces but rather their disjoint union.

4.3 Specific Lingua Franca Conformance Rules

As mentioned already, we have chosen to use structural rather than name based conformance rules. Roughly speaking, if one type has "at least" as much as another type, the former conforms to the later. For example consider the following types *Sub* and *Super*

```
type Super = record of
                s : String;
                i : Integer;
             end record;
type Sub =   record of
                i : Integer;
                s : String;
                b : Boolean;
             end record;
```

Since all the fields of type *Super* are in type *Sub*, it is easy to see that a value of type *Sub* can be used anywhere a value of type *Super* is expected. In this case, *Sub* conforms to *Super*.

Formally in our system, a type α conforms to type β if and only if one of the following is true:

1. β and α are interface types and for each method μ_β of β there exist a method μ_α of α such that the following conditions hold:
   (a) μ_α and μ_β have the same name and number of parameters
   (b) the parameters of μ_β conform to the parameters μ_α
   (c) the result of μ_α conforms to the result of μ_β
2. α and β are primitive types and α equals β
3. α is the type "sequence of ϵ_α" and β is the type "sequence of ϵ_β" and ϵ_α conforms to ϵ_β
4. α is the type "sequence of N_α ϵ_α" and β is the type "sequence of N_β ϵ_β" and ϵ_α conforms to ϵ_β and $N_\alpha = N_\beta$
5. α is the type "sequence of N_α ϵ_α" and β is the type "sequence of ϵ_β" and ϵ_α conforms to ϵ_β
6. α is the type "pointer to ρ_α" and β is the type "pointer to ρ_β" and ρ_α conforms to ρ_β
7. α and β are case types and for each field ϕ_α of α there exist a field ϕ_β of β such that the following conditions hold:
   (a) ϕ_α and ϕ_β have the same name
   (b) the type of ϕ_α conforms to the type of ϕ_β
8. α and β are record types and for each field ϕ_β of β there exist a field ϕ_α of α such that the following conditions hold:
   (a) ϕ_α and ϕ_β have the same name
   (b) the type of ϕ_α conforms to the type of ϕ_β
9. β is the *Anything* type (i.e. all types conform to *Anything*)

Our rules for conformance assume that arguments are passed without side effects. This is true given our chosen semantics for argument passing. For this reason, it is only necessary to test the contravariance relationship for arguments. If, at a latter stage, the language is extended to support pass-by-reference or pass-by-value-return then the conformance rules will have to be modified so that the the covariance relationship for arguments with side-effects is tested as well.

The rules for conformance are recursive and when applied to recursive types equate to testing that the infinite expansions of the types structurally conform. Unfortunately, this implies the rules are not finitary. However, finitary algorithms exist that test the rules of structural conformance even in the presence of recursion[2]. To understand how this works, we first define the representation of a type to be a directed graph where the nodes of the graph represent types and edges are pointers to types used in the source node's construction. The nodes contain the necessary information to perform the conformance test. E.g., a structure node contains a name for each outgoing edge. Every type can be represented as a graph with a finite number of nodes. Cycles in

the graph indicate recursion. The algorithm to test structural conformance takes two nodes in the graph and returns TRUE or FALSE whether the first argument structurally conforms to the second argument. A high-level outline for the algorithm is:

1. If F conforms to T then return TRUE.
2. Assume F conforms to T. Future recursive calls to the algorithm with (F,T) as the argument will return TRUE immediately in Step 1.
3. Verify F conforms to T by testing the conformances rules using the data within the nodes and return either TRUE or FALSE. This checking may make recursive calls to this algorithm but recursive calls with (F,T) as the argument will return immediately with TRUE.

A detailed look at an equivalent algorithm is provided in [2]. The algorithm guarantees the conformance rules are never verified for a pair of types more than once. Since the number of nodes in a type graph is finite, the number of verifications of the conformance rules are finite, and the algorithm will terminate even in the presence of recursion. In the absence of recursion steps 1 and 2 of the algorithm are not necessary.

4.4 Comparison of Interface and Structural Subtyping

The problem with inheritance subtyping is that it increases coupling between clients and servers. Clients must specify their requirements as a named interface and can only inter-operate with server objects that are in a inheritance relationship with this interface. Often, a client may have to over specify its actual requirements resulting in a situation where changes to an object's interface unduely effect the inter-operability between the client and server. The following is an example of this problem which we term the *false interface dependency* problem:

```
type VarOpaque = sequence of Byte;
type File = interface of
    read( Integer ) : VarOpaque;
    write( Integer, VarOpaque )
        : Integer;
    length() : Integer;
    kind() : case of
        FILE: Nil;
        DIRECTORY : Nil;
    end case;
end interface;
```

```
NewFile = interface of
    read( Integer ) : VarOpaque;
    write( Integer, VarOpaque )
        : Integer;
    length() : Integer;
    kind() : case of
        FILE: Nil;
        DIRECTORY : Nil;
        PIPE : Nil;
    end case;
end interface;
```

In this example a file system has been extended to support named pipes. Unfortunately, **NewFile** cannot be placed in a subclass relationship with **File**. Without going into a rigorous proof, consider what might happen if **NewFile** were a subclass of **File**. Clients expecting an object of type **File** could be given objects of type **NewFile**. This leads to type errors since clients only expect the kind() operation to return either FILE or DIRECTORY and invocations returning PIPE would not be expected. Consequently, clients written to use objects of type **File** can not inter-operate with servers exporting objects of type **NewFile**. This is true even if the actual requirements of the client would allow it to inter-operate with both **File** and **NewFile**. For example, a print server that takes a **File** object for printing may actually only require the read and length operation but, because the print server over specified its requirements as taking a **File** object, it cannot inter-operate with objects of type **NewFile**:

```
type PrintServer = interface of
    print( File ) : Boolean;
end interface;
```

The **File** and **NewFile** interfaces can be used to illustrate another problem; that the compatability between clients and servers may be falsely dependent upon the inheritance hierarchy. For example, **NewFile** could not be made a subclass of **File**, but the converse is not true; **File** could be made a subclass of **NewFile**. This would allow clients using the **NewFile** interface to inter-operate with objects of type **File**. However, this requires retroactively modifying the hierarchy which is not practical since it is globally shared among cooperating domains. Consequently, even if the print server is rewritten to accept objects of type **NewFile**, it will not inter-operate with objects of type **File**. We term this second problem the *false hierarchy dependency problem.*

There is a subtle difference between the two problems. In the false interface dependency problem, interfaces can not be placed in an inheritance relationship. In the false hierarchy dependency problem, it

is not that the interfaces can not be placed in an inheritance relationship, but, rather they are not for what ever reason. By not placing interfaces in an inheritance relationship, clients may be unnecessarily incompatable with servers.

Both of these problems are solved trivially using structural subtyping. The print server need only specify its requires as a narrower view of `File` that objects of both type `File` and type `NewFile` conform to:

```
type VarOpaque = sequence of Byte;
type PrintServerFile = interface of
    read( Integer ) : VarOpaque;
    length() : Integer;
end interface;
type PrintServer = interface of
    print( PrintServerFile )
        : Boolean;
end interface;
```

Using structural subtyping the print server will work with both `File` and `NewFile` objects.

Appendix A further illustrates the differences between structural subtyping and interface inheritance by discussing a scheme to approximate structural subtyping with interface inheritance. The net result is that interface subtyping is only minimally effective at achieving the expressive power of structural subtyping especially in the presence of recursive types and contravariance of arguments.

4.5 Problems with Structural Subtyping

One possible disadvantage to using structural subtyping is the loss of the semantic information that hierarchies of interface names provide. We address the problem in *Lingua Franca* by supporting the augmentation of objects with semantic attributes. Semantic attributes are treated as type predicates on the object. For example, `sorted` or `FIFO`. In addition to supplying an interface when narrowing an object reference, a client may supply a list of semantic attributes for checking semantic conformance. For example, a client's interface to a file system might contain `read` and `write` methods. Many different file systems types could support this as a subset of their interface. For example, `ReplicatedFile`, `CompressedFile`, `EncryptedFile`. Clients which do not care which type of file they use, can simply narrow references they obtain to their notion of a file. An application which needs encrypted files, can specify the `encrypted` attribute during the narrowing (assuming the `EncryptedFile` objects are the only type of file objects that possess this semantic attribute).

The argument might be made that the different file types would also have additional methods. For example, `EncryptedFile` might provide a `password` operation. These additional methods could also be used by the client to distinguish between the different types of files. A client interested in an encrypted file could specify this by including the `password` operation in the interface used during narrowing. This technique actually implies the way in which semantic attributes are implemented: as nullary methods automatically added to the client's interface during narrowing and to the interface of objects specified to possess them. Attributes are still useful when the semantic information can not be inferred from the interface. For instance, consider the interfaces `stack` and `queue` that both have the same set of methods: `add` and `remove`. In this case, where a client is unable to distinguish between a stack and queue object based interface alone, the semantic attributes `LIFO` and `FIFO` could be used.

Of course attributes are not necessary at all. As eluded to earlier, attributes can be implemented as nullary methods on interfaces (e.g. an `ImAFIFO()` method). However, it is our opinion that the semantic information of an object and its interface are conceptually different and a clear distinction between them should be made.

5 Performance

A complete analysis of the performance of our system is beyond the scope of this paper and is detailed elsewhere[17]. However, it is interesting to look at two facets of the system to put it in perspective with other work.

5.1 Performance Of Cross Domain Method Invocation

Figure 5 shows the method invocation times for a null method (no arguments/no results) for various combinations of *Renaissance* clients and servers. These data and all the following were gathered on SPARCStation IPX's and SPARCStations 10's with 32Mb of memory on an otherwise idle Ethernet. The test includes using a *Renaissance* kernel resident server, a user-level application running on that kernel and a UNIX user-level application server, as well as clients in all three of these same domains. UNIX clients and severs were run on different machines, the kernel and its applications were run on

Clients	*Server* Kernel	Application	UNIX
Kernel	0.29μs	501μs	1999μs
Application	51μs	626μs	1959μs
UNIX	2544μs	2295μs	3137μs

Figure 5: Method Invocation Times for Various Client/Server Domain Combinations

	Kernel	Application	UNIX
1	16μs	414μs	2914μs
2	29μs	437μs	2970μs
3	45μs	460μs	2970μs
4	62μs	460μs	2976μs
5	81μs	487μs	3020μs

Figure 6: Reference Narrowing Times for a Kernel Server Object with 5 Methods

the same machine.

The affect of our support for customized transports in the runtime support libraries is evident in the speed of the *Renaissance* kernel to kernel method invocation times (it is optimized to a C++ virtual function call) and the application to kernel time (it is optimized to used shared memory). The remaining cases use UDP/IP as a transport and also pay the cost of data canonicalization.

5.2 Performance of Run-Time Conformance Checking

Figures 6 and 7 show the expense of narrowing a reference to a client's interface for various types of client applications contacting a kernel server and a UNIX server respectively. These number were gathered with the conformance checking cache disabled, and therefore these times reflect the actual time to run the conformance checking algorithm, including transport overheads. The leftmost column lists the number of methods in the client interface. All the methods in all the interfaces were null methods, so these times represent the minimum cost of narrowing references in each case. Argument/result conformance checking will add to the expense depending on the complexity of the types involved.

	Kernel	Application	UNIX
1	2503μs	2585μs	4282μs
2	2474μs	2542μs	4170μs
3	2503μs	2584μs	4164μs
4	2519μs	2572μs	4470μs
5	2536μs	2642μs	4218μs

Figure 7: Reference Narrowing Times for a UNIX Server Object with 5 Methods

6 Related Work

Clients in traditional distributed systems such as V [8, 7], Mach [1], and Chorus [21] acquire system services from servers by explicitly sending messages to ports or processes. While the servers in such systems can be considered objects and message sending analogous to invoking methods, such systems do not provide a run-time representation for types and therefore cannot perform conformance checking. Also, in these systems the burden of data canonicalization and type checking is placed on the programmer.

Our approach is obviously related in overall philosophy to a number of other systems that use an interface description language. Object-based systems including OMG's CORBA[19], Microsoft's OLE2[15], IBM's SOM/DSOM[12] and Sun's Spring system[11] all provide an IDL for describing objects and translators for mapping these descriptions into target languages. Likewise, remote procedure call[5] (RPC) based systems such as SUN RPC[24] provide an IDL for describing services but in terms of procedures not objects. Clients in RPC-based systems acquire services by invoking local functions that transparently access remote services. Most RPC-based systems provide the notion of a program and a set of procedures to call within the program and is analogous to an object with methods. Although, object-based and RPC-based systems appear very similar, there are two important differences. First, unlike object-based systems where interfaces are first class types, most RPC-based systems do not support procedures (or programs) as first class types. Secondly, RPC-based systems use name equivalence for conformance checking and do not have a hierarchy of type names. What separates our work from the others systems is the use of an IDL that uses structural subtyping rather than names of interfaces for conformance testing.

Distributed object languages, such as Argus [14], Distributed Smalltalk [3], and Emerald [20], and distributed object systems, such as Clouds [9] and

Eden [13], not only provide a notion of objects and type conformance but also provide features such as concurrency and atomicity, replication, persistence, fault tolerance, and migration. Unfortunately, the requirements placed on these systems to support these features makes it difficult for them to scale and interoperate with one another. In our system, such features are considered object-specific and would be provided by the implementation of the object and hidden from the client behind the object's interface. We emphasize the accessing of remote objects independent of their particular implementation and provide a framework in which systems can interoperate.

7 Conclusion

We believe the use of structural subtyping to be superior in achieving the high degree of autonomy between domains and the low coupling between software components necessary for a scalable, extensible distributed system. With the structural subtyping our system provides there are no global type hierarchies to share among domains, but rather, all objects are self describing. These descriptions, available to the programmer via the signature objects, are used for conformance testing and documentation. The lack of a global type hierarchy allows new service types to be easily added to the system and existing service types to be easily modified.

The structural subtyping conformance rules we have chosen are more flexible and natural than inheritance based name conformance rules. Conformance tests that fail using inheritance may succeed using structural subtyping. Also, programmers can make use of the flexible and dynamic nature of our conformance test. For instance, an object, for security purposes, can selectively export different subsets of methods by "pretending" to have different interfaces during the narrowing process. This subset may be determined at run-time based on the client's identity.

The self-describing nature of our system has several advantages. First, the distributed environment is more manageable. As software evolves and clients become incompatible with servers, the run-time type information can be used to determine how they are incompatible and what changes need to be made to the client. The run-time type information also provides on-line documentation about an object's interface and can aid in resource discovery. Second, our system supports statically and dynamically typed programming equally well. A C++ translator for *Lingua Franca* has been built that maps *Lingua Franca* type descriptions into C++ types. Once an object is narrowed, the static type checking of C++ guarantees its proper use. Also, a World Wide Web based object browser/invoker was easily constructed by mapping dynamically discovered object interfaces into HTML documents and constructing a HTML form based translator that invokes dynamically discoved interface methods.

Using structural subtyping for run-time conformance checking is not without cost. The run-time type information imposes some size penalty on individual objects, but much of this information can be shared between objects of the same type. The greatest expense lies in the run-time costs of the conformance checking algorithm that were discussed in Section 5.2.

Our work provides programmers with an object-oriented framework within which to construct distributed object systems. In addition, it brings a language-independent notion of object and type (interface) closer to reality. Finally, providing the ability to separate implementation details from interface details, and semantic information from syntactic, allows the construction of less coupled, more extensible distributed software.

Appendix A: Approximating Structural Subtyping Using Interface Inheritance

To further illustrate the difference between structural subtyping and interface inheritance is interesting to try to approximate structural subtyping with interface inheritance. For this section we will use the following symbols:

- $I_x :< I_y$ if and only if I_x structurally conforms to I_y
- $I_x = I_y$ if and only if $I_x :< I_y$ and $I_y :< I_x$ (this is also termed *structural equivalence*)
- $I_x < I_y$ if and only if I_x is a subclass of I_y (i.e. I_x inherits from I_y)

To approximate structural subtyping for an interface I_s we mean that for each interface I_c where $I_s :< I_c$ there exist an interface $I_{c'}$ such that $I_s < I_{c'}$ and $I_c = I_{c'}$. I.e., for each client requirement that the server structurally conforms to there must exist an interface in the hierarchy equivalent to it. We assume the interface hierarchy is comprised of a finite set of interfaces and therefore the best we can

hope to do is approximate a finite set of client requirements. However, it can be shown that there are potentially an infinite number of distinct client interfaces that a server conforms to[18]. Two interfaces are distinct if they are not structurally equivalent.

The general approach is to enumerate all possible structural supertypes (client requirements) of an interface and use multiple inheritance to build up the interface hierarchy. To guarantee that the proper structural subtyping relationships are approximated in the resulting hierarchy, the following invariant should hold: *if an interface is a structural subtype of another interface then it must be a subclass of the interface.*

The scheme we use for the approximation takes each interface I_s and gives a name to all possible combinations of the methods of I_s. This is done by creating a separate interface for each method in I_s and using multiply inheritance to create the possible combinations of them. For example consider the following interfaces `A` and `B`:

```
type B = interface of
    b1() : Nil;
    b2() : Ni;
end interface;
type A = interface of
    a1() : B;
    a2() : Nil;
    a3() : Nil;
end interface;
```

The resulting interface hierarchy for `A` and `B` would be:

```
type B_b1 = interface of
    b1() : C;
end interface;
type B_b2 = interface of
    b2() : C;
end interface;
type B = B_b1 + B_b2;
type A_a1 = interface of
    a1() : B;
end interface
type A_a2 = interface of
    a2() : Nil;
end interface
type A_a3 = interface of
    a3() : Nil;
end interface
type A_a1_a2 = A_a1 + A_a2;
type A_a1_a3 = A_a1 + A_a3;
type A_a2_a3 = A_a2 + A_a3;
type A = A_a1_a2 + A_a1_a3 +
         A_a2_a3;
```

Clients can now specify their requirements as the precise set of methods needed for `A` and `B`. Care must be taken when defining the interfaces and their inheritance relationships so that invariant holds. If `A` were defined as:

```
type File = A_a1 + A_a2 + A_a3;
```

then `A` would not inherit from `A_a1_a2`, the invariant would not hold, and objects of type `A` could not be used where objects of type `A_a1_a2` are expected.

This approximation scheme has severe limitations. It does not handle contravariance of arguments or covariance of results. Contravariance allows clients to over specify the requirements of an argument. E.g., there are an infinite number of distinct interfaces for this over specification. Approximating contravariance requires giving a name to each of these possible interfaces which violates the restriction that a finite number of interfaces be used in the approximation. Covariance allows clients to under specify their requirements. By re-appling our approximation scheme to the results of the method, it is possible to approximate covariance. For each method, a set of methods is created one for each possible supertype of the result. Each of these are placed in a separate interface and multiple inheritance is used to create the proper structural subtyping relationship. Reconsidering the interfaces `A` and `B` again, and applying the scheme to the results of methods, yields the following changes to the previous hierarchy:

```
type A_a1_b1 = interface of
    a1() : B_b1;
end interface
type A_a1_b2 = interface of
    a1() : B_b2;
end interface
type A_a1 = A_a1_b1 + A_a1_b2 +
            interface of
                a1() : B;
            end interface;
type A_a2 = interface of
    a2() : Nil;
end interface
type A_a3 = interface of
    a3() : Nil;
end interface
type A_a1_b1_a2 = A_a1_b1 +
                  A_a2;
type A_a1_b1_a3 = A_a1_b1 +
                  A_a3;
type A_a1_b2_a2 = A_a1_b2 +
                  A_a2;
type A_a1_b2_a3 = A_a1_b2 +
```

```
                    A_a3;
type A_a1_a2 = A_a1 + A_a2 +
               A_a1_b1_a2 +
               A_a1_b2_a2;
type A_a1_a3 = A_a1 + A_a3 +
               A_a1_b1_a3 +
               A_a1_b2_a3;
type A_a2_a3 = A_a2 + A_a3;
type A = A_a1_a2 + A_a1_a3 +
         A_a2_a3;
```

A graphical picture of the resulting hierarchy is shown in Figure 8. The solid lines and ovals are from appling our scheme to A, the dashed lines and ovals are from re-appling our scheme a second time to the results of the hierarchy. One thing that is immediately apparent is the relatively complicated hierarchy that results from such a seemingly innocent problem. The solution suffers from the need to pollute the namespace with the addition interface names and a complex inheritance scheme.

This scheme can iteratively be applied to the resulting hierarchy until all supertypes of an interface are enumerated. However, there is a serious short coming to this approach: for recursive interfaces the scheme can potentially be applied infinitely many times. It can be shown that recursive interface potentially have an infinite number of distinct client interfaces it structurally conforms to, but only a finite number of these client requirements (the ones named) can be approximated. For example consider the following recursive interfaces:

```
type Directory = interface of
    openDirectory( String )
        : pointer to Directory;
    list() : sequence of String;
    ...
end interface;

type NewDirectory = interface of
    openDirectory( String )
        : pointer to NewDirectory;
    list() : sequence of String;
    ...
end interface;

type ClientDirectory = interface of
    openDirectory( String )
      : pointer to ClientDirectory;
    list() : sequence of String;
end interface;
```

Assume NewDirectory cannot be placed in a subclass relationship with Directory and the client requirements are specified by ClientDirectory. Applying our scheme a finite number of times will not yield an interface hierarchy that approximates the ClientDirectory interface. However, both Directory and NewDirectory are structural subtypes of ClientDirectory.

We have presented only one approximation scheme. One that fully approximates non-recursive types when contravariance is ignored. This is possible because all possible structural supertypes of a non-recursive type can be enumerated (ignoring contravariance). There are others, but they all have the same general form. A finite set of client requirements are enumerated and interface inheritance is used to establish the proper structural subtyping relationships between them. Consequently, they all have the same problems and limitations: they clutter the namespace, they add complexity to the hierarchy, and they fail to fully approximate interfaces in the presence of recursion. For this reason, these approaches are of mostly theoretical interest. In practice, the programmer of the interface decides which client requirements to support usually by subclassing from a small number of existing interfaces or by creating a small number of new interfaces. The problem with this is the programmer of the *server* determines what are the possible *client* requirements.

References

[1] Mike Accetta et al. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX Conference*, pages 93–111, June 1986.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping Recursive Types. *ACM Transactions on Programming Languages and Systems*, 14(4):575–631, 1993.

[3] John K. Bennett. The Design and Implementation of Distributed Smalltalk. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 318–330, 1987.

[4] Edward V. Berard. *Essays on Object-Oriented Software Engineering*, volume 1. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[5] Andrew Birrell and Bruce Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

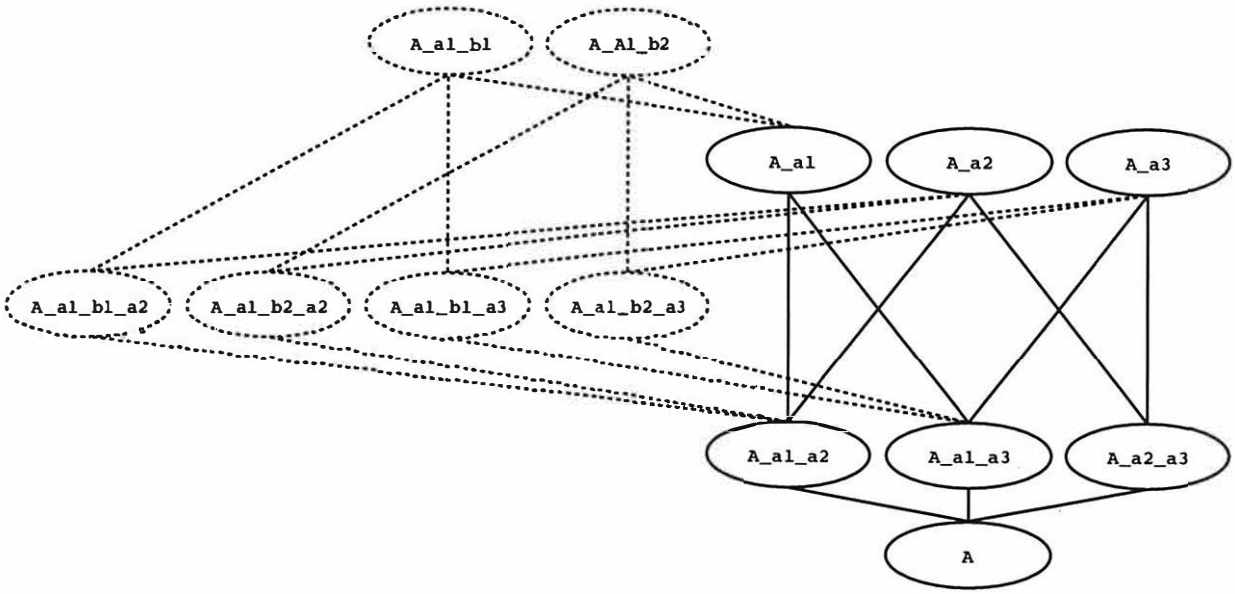


Figure 8: A Multiple Inheritance Hierarchy for interface A

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[7] David R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–42, April 1984.

[8] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[9] Partha Dasgupta, Richard LeBlanc, and William Appelbe. The Clouds Distributed Operating System:Functional Description, Implementation Details, and Related Work. Technical Report GIT-ICS-87/42Functional Description, Implementation Details, and Related Work, Georgia Tech, 87.

[10] Carlo Ghezze, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[11] G. Hamilton, M. Powell, and J. Mitchell. Subcontract: A Flexible Base for Distributed Programming. In *Proceedings of the ACM Symposium on Operating System Principles*, December 1993.

[12] IBM. *SOMobjects Developer Toolkit Users Guide Version 2.0*, 1993.

[13] E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, and S. Vestal. The Architecture of the Eden System. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 148–159, 1981.

[14] Barbara Liskov. Distributed Programming in Argus. Technical Report Programming Methodology Group Memo 58, MIT, October 1987.

[15] Microsoft Corporation. *OLE2 Programmer's Reference*, volume 2. Microsoft Press, 1994.

[16] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.

[17] Patrick A. Muckelbauer and Vincent F. Russo. Efficient Remote Method Invocation in a System Utilizing Structural Subtyping. Technical report, Department of Computer Sciences, Purdue University, June 1995.

[18] Patrick A. Muckelbauer and Vincent F. Russo. Lingua Franca: an IDL for Structurally Subtyping Distributed Systems. Technical report, Department of Computer Sciences, Purdue University, June 1995. This is and expanded form of the current paper.

[19] OMG. *The Common Object Request Broker: Architecture and Specification*, 1991.

[20] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchison, and Eric Jul. Emerald: A General-Purpose Programming Language. *Software – Practice and Experience*, 2(1):91–118, January 1991.

[21] M. Rozier, V. Abrossimov, and W Neuhauser. CHORUS-V3 Kernel Specification and Interface, Draft. Technical Report CS/TN-87-25.10, CHORUS Systems, February 1988.

[22] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th. International Conference on Distributed Computer Systems*, May 1986.

[23] Bjarne Stroustrup. *The C++Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[24] Sun Microsystems. *Networking on the SUN Workstation*, 1985.

[25] Guido van Rossum. *Python Reference Manual*. CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1994.

Adding Group Communication and Fault-Tolerance to CORBA*

Silvano Maffeis
maffeis@acm.org
Department of Computer Science
Cornell University, Ithaca, NY

Abstract

Groupware and fault-tolerant distributed systems stimulate the need for structuring activities around object-groups and reliable multicast communication. The object-group abstraction permits to treat a collection of network-objects as if they were a single object; clients can invoke operations on object-groups without needing to know the exact membership of the group. Object-groups mainly serve to increase reliability through replication, performance through parallelism, or to distribute data from one sender to a large number of receivers efficiently. This paper describes how object-groups and reliable multicast communication can be added to a CORBA compliant Object Request Broker. It also presents ELECTRA — a CORBA Object Request Broker whose architecture is pervaded by the group concept.

Keywords: Object-Groups, Multicast, Replication, CORBA, Electra, Horus, Isis

1 Statement of Problem

1.1 One World: CORBA

Object-oriented programming is believed to be one of today's best programming models to cope with complex systems while providing maintainability, extensibility, and reusability [14]. A model claiming such attributes is particularly interesting for distributed systems, as these tend to become very complex.

In industry, the client-server model is finding increasing consideration for interconnectivity. In this model, servers provide clients with access to services such as file storage or authentication by IPC mechanisms like message passing or RPC. Object-oriented, *distributed* programming (OODP) is a generalization of the client-server model, in that objects encapsulate an internal state and make it accessible through a well-defined interface. Client applications may import an interface, bind to a remote instance of it, and issue remote object invocations [6]. This use of objects naturally accommodates heterogeneity and autonomy: heterogeneity since messages sent to objects depend only on their interfaces and not on their internals, autonomy because object implementations can change transparently, provided they maintain their interfaces [16].

In 1989, the Object Management Group (OMG) started elaborating an open standard for OODP, called the Common Object Request Broker Architecture (CORBA) [18]. At the time of this writing, OMG counted more than 500 members worldwide, including enterprises like Apple, DEC, HP, IBM, NCR, Novell, and SUN. Many of the large software producing firms have committed to make their products comply with CORBA withing the next few years.

People living in what we could call the "CORBA world" emphasize aspects such as reusability, portability, interoperability, and integration of legacy information systems. Unfortunately, the current version of CORBA as well as the Object Request Brokers (ORBs) in use today do not adequately treat two of the most fundamental problems which occur in real-world distributed applications, namely partial failures and consistent ordering of distributed events [9]. In face of partial failures, large distributed applications implemented with current ORB technology may behave in an unpredictable way, leading to inconsistent data or to other malfunctions. Moreover, reliable asynchronous communication is not adequately supported in the CORBA standard, though asynchronous communication is important for building scalable distributed applications, as it permits to overlap computation with communication and to hide network latencies.

*Research supported by grants from the Swiss National Science Foundation, Siemens-Nixdorf, Union Bank of Switzerland, and KWF/CERS

1.2 Another World: Horus, Isis, etc.

In current distributed systems research, there is a growing community working on problems related to partial failures and on system models to ensure predictable behavior of distributed applications. Examples of such models are "Virtual Synchrony" [3], "Pseudo Synchrony" [19], and "View-Synchronous Communication" [1, 20]. The models are similar; they mainly ensure that related actions taken by different processes of a distributed system are mutually consistent, and that actions are timely correct even when multiple components communicate asynchronously to perform some task. Hence, by following these models the behavior of a distributed application is predictable despite crashes, asynchronous communication, and in spite of processes joining and leaving the system dynamically.

Process-groups and fault-tolerant multicast are at the heart of the models. The models mainly differ in how membership-changes are propagated to the members of a process group, in how partitioned networks are treated, and in how context information associated with messages is represented.

In contrast to CORBA, these models address issues related to partial failures, process replication, reliable multicast, asynchronous communication, and ordering of events. Implementation of robust distributed systems on conventional hardware is enabled by toolkits that follow these models, e.g., by Horus [25], Isis [3], Transis [1], and Consul [15]. Unfortunately, the programming interface provided by these toolkits is proprietary and rather low-level; it mainly consists of a rich set of C-procedures presenting access to light-weight processes, unstructured messages, process groups, message passing primitives, and so forth. Moreover, it is hard to port an application from one toolkit to another.

1.3 Electra: Bridging the Gap

Indeed, the two worlds provide complementary functionality and are both very important for future distributed systems. The goal of our work has been to design and implement Electra — a novel programming environment combining the benefits of CORBA with the strengths of systems like Horus and Isis. Electra is a CORBA-compliant ORB which, in addition to the functionality provided by today's ORBs, permits the grouping of object-implementations, reliable multicast communication, and object-replication. Electra is conceived to run on platforms such as Horus and Isis. We believe that an ORB should be based on primitives as provided by Horus or by one of the other aforementioned toolkits, and not directly on the primitives of contemporary operating systems, e.g., RPC, UNIX sockets, or Windows NT pipes. In addition, a flexible system design has been devised allowing developers to customize Electra for various toolkits. Hence, Electra can be thought of as a *generic* Object Request Broker.

Details of the Electra object model are provided in [13], whereas in [12] the performance of Electra is assessed. The rest of this paper is structured as follows. In the next section we describe the object model underlying our ORB. Group-communication is addressed in Section 3, whereas Section 4 describes enhancements we made to the standard CORBA interfaces in order to support group-communication and fault-tolerance. Section 5 deals with the mapping of CORBA interfaces onto object-references that support both unicast and group-communication. Portability of Electra is addressed in Section 6. Finally, Section 7 compares Electra with conventional ORBs and concludes the paper.

2 The Electra Object Model

In our work we consider *asynchronous distributed systems* consisting of objects which run on a collection of machines, and which interact by message passing or remote method invocation. In asynchronous systems, there are no constraints on the speed at which objects make progress and on the message transmission delays. Furthermore, neither an exact synchronization of the local clocks nor a reasoning based on "global time" [9] is possible. In this situation, interprocess communication remains the only feasible means of synchronization. We further assume that failures respect the *fail-stop* model [21], which means that objects fail by crashing without the emission of spurious messages.

In analogy to the OMG Object Management Architecture (OMA) [23], the Electra object model consists of objects implemented in various programming languages scattered over an arbitrary number of machines. The ORB is the communication heart in the model. It provides an infrastructure allowing objects to communicate, independent of the specific programming languages and techniques used to implement the objects. Client applications use object-references to send messages to remote object-implementations. References are valid across node boundaries and can thus be passed from one node to another. Furthermore, objects are not tied to a client or server role; a client acting as server at a certain moment can be-

come a client at a later point and vice versa.

Electra enhances OMA in that objects can be aggregated to so-called *object-groups* [11]. An object-group is a means of combining network-objects and naming them as a unit (Figure 1). Communication is by *reliable multicast*, which means that a CORBA operation issued through a group-reference is received by all implementations which are members of the group.

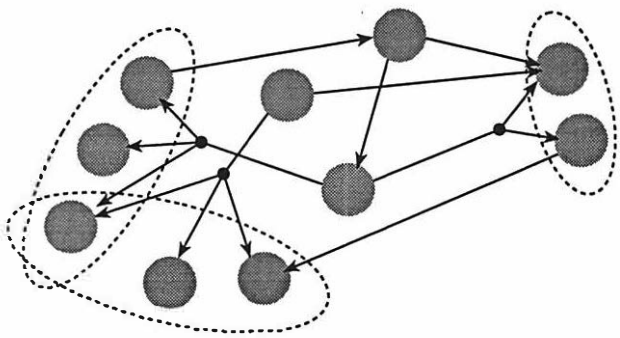


Figure 1: Electra objects communicating by point-to-point and multicast invocations. Circles represent objects, dotted ovals define object-groups.

Programmers can bind object-references to both singleton objects and object-groups using the same programmatical expressions, and multicasts can be issued both through the CORBA static and dynamic invocation interface. Object-group communication can be performed in a transparent or in a non-transparent way. In transparent mode, an object-group appears as if it was a highly available singleton object. In contrast, non-transparent communication permits programmers to access the results of an invocation which were produced by the individual group members.

Another difference to CORBA is that object invocations can be performed synchronously, asynchronously, or deferred-synchronously, through both the static and dynamic invocation interface, and both for singleton and group destinations. Within an object-implementation, each invocation obtains its own thread of execution. The Electra model is thus inherently asynchronous and multi-threaded.

Applications fitting our model are such requiring efficient multicast, asynchronous communication, fault-tolerance, or a combination thereof. Prospective application areas of Electra are video-on-demand servers, video conferencing systems, groupware, distributed parallel computing, and different kinds of fault-tolerant client-server applications.

3 Object-Groups

3.1 Motivation

In distributed systems, communication can take two different forms depending on the number of participants: point-to-point and multicast. The first form is trivial and is provided by conventional communication mechanisms like message passing or RPC. The second form is more powerful and more complicated than point-to-point communication, it has recently received much attention [3, 7, 27, 4, 8]. *We believe that the conjunction of the group communication model with the object model will lead to a compelling programming paradigm for future distributed systems.*

If the underlying communication hardware or software provides a multicast facility, as it is the case in the Ethernet or in an extended version of the IP protocol (so-called IP-Multicast or MBONE [2]), Electra can take advantage of it to transmit group invocations efficiently. In the worst case, a multicast is mapped into one point-to-point message per group member. This affects the performance of an Electra application but not the programming model.

3.2 Reliability

If Electra is configured to run on one of the toolkits we mentioned in Section 1.2, object-group communication is reliable. This means that when an operation is issued through a group-reference, all operational group members will dispatch the same set of operations (Agreement), that this set will include all operations multicast by operational objects to the group (Validity), and that no spurious operations will ever be dispatched (Integrity) [5]. Low-level reliable multicast is realized by the underlying toolkit and not by Electra. Electra's contribution is in providing an easy-to-use, CORBA-compliant interface to group communication.

3.3 Ordering of Events

Creating object-groups as well as joining and removing objects from groups is accomplished by special Electra operations which were included into the CORBA Basic Object Adapter (BOA). When creating an object-group, programmers specify ordering requirements for the invocations dispatched by the group members. The ordering protocols available depend on the underlying toolkit. For instance, if Electra is configured for Isis, programmers can specify that all group members dispatch all operations in ex-

actly the same order (Isis *abcast* protocol), in a causal order [9] (Isis *cbcast* order), or by Isis *gbcast* protocol [3]. If the programmer specifies an ordering-protocol that is not available in the underlying toolkit, an exception is thrown.

3.4 Application Areas

Object-groups have many interesting application areas:

- **Fault-Tolerance**: Availability and fault-tolerance of an object are increased by using active replication, passive replication, or multi-versioning. Each approach can be mapped onto object-groups. In Electra, an object fails independently of the other members[1], and the service remains available as long as at least one of the members is operational.
- **Load Sharing**: A singleton object can be replaced by an object-group when the object becomes overloaded, and, in many cases, without having to modify applications which use the object. For instance, the group members may share the available work-load to increase throughput. Parallelism and load sharing can thus be increased step by step.
- **Caching**: In certain situations, the response time of a service is decreased if provided by an object-group, since member-objects can be placed at the sites where the service is frequently accessed.
- **Efficient Data Distribution**: Object-group multicast can be mapped onto hardware (e.g., Ethernet) or software (e.g., MBONE [2]) multicast facilities. This allows the same network-message to be received by all members of an object-group.
- **Network Management**: Object-groups offer a convenient solution to the problem of propagating monitoring and management information in distributed applications.
- **Mobility**: In Electra, multicasts are issued through opaque CORBA object references and senders do not need to know the network addresses of the members. Consequently, an object can leave a group, move to another place, then rejoin the group and continue working. Object-groups hence offer support for mobility and for system reconfiguration.

[1] provided that the group members were instantiated on different machines.

4 The BOA and Environment Class

Electra is implemented in the C++ programming language and C++ is the only target-language supported by the present version[2]. Our IDL-to-C++ mapping follows the specification in OMG Document 94-9-14 [17]. To add group-communication and fault-tolerance to CORBA, only two C++ interfaces had to be enhanced with a few special operations: operations for managing object-groups were included in the `BOA` class, while operations for selecting an invocation style were added to the `Environment` class. Note that both classes still comply with the CORBA standard, since new methods were added to them without altering signature or semantics of the standard methods.

4.1 Enhanced BOA Interface

In Electra, an object-implementation is an instance of a subclass of the `BOA` class below. Thus, `BOA` operations can be issued on any object-implementation.

```
// C++
class BOA {
public:
    // Standard BOA interface. See OMG doc. 94-9-14:
    //
    Object_ptr create(const ReferenceData&,
        InterfaceDef_ptr, ImplementationDef_ptr);
    void dispose(Object_ptr);
    ...

    // Electra-specific operations:
    //
    static void create_group(Object_ptr group,
        const ProtocolPolicy& policy
        =default_protocol_policy,
        Environment_ptr =0);
    void join(Object_ptr group, Environment_ptr =0);
    void leave(Object_ptr group, Environment_ptr =0);
    static void destroy_group(Object_ptr group,
        Environment_ptr =0);

    virtual void get_state(AnySeq& state,
        Boolean& done, Environment_ptr env);
    virtual void set_state(const AnySeq& state,
        Boolean done, Environment_ptr env);
    virtual void view_change(const View& newView);
};
```

The `BOA::create_group` method creates a new object-group and binds the object-reference **group** to it. The reference can be installed in a name server or converted to a human-readable string by the `ORB::object_to_string` operation. The **policy**

[2] language bindings for Smalltalk, Lisp, Fortran, SML etc. can be provided by writing language-specific backends for the IDL compiler.

argument is used to tell the underlying toolkit what kind of multicast protocol to employ, e.g., for total ordering or causal ordering [9, 5]. If Electra is configured to run on Isis, the `policy` object will select either the Isis *abcast*, *cbcast*, or *gbcast* protocol to transmit a multicast. In the Horus configuration, the `policy` object serves to compose a Horus protocol stack [26]. For instance, the programmer can specify ATM as transport layer and pick from a variety of ordering protocols to be placed atop of the ATM layer. The policy-object mechanism could be extended to cover quality of service guarantees. For example, the minimum bandwidth necessary to sustain a certain service could be defined through a policy object.

Objects in the network join or leave a group simply by retrieving its reference from the name server and by issuing the `join` or `leave` operation with the reference as parameter. `destroy_group` irrevocably destroys an object-group. Note that the group-members themselves are not destroyed.

When an object joins a non-empty group, Electra will obtain the internal state (i.e., the values of all instance variables) of some group-member by invoking its `get_state` method. Subsequently, Electra transfers the state to the newcomer and invokes the newcomer's `set_state` method. A large state can be transferred in fragments. For this purpose, Electra will continue to invoke the state transfer methods until `TRUE` is assigned to the `done` return argument of `get_state`. The `Environment` object is used to signal an interrupted state transfer due to a failure of the member from which the state was being received. An object-state is represented as a sequence of CORBA any objects.

State transfer is necessary for redundant computations to permit the replication-degree of an object to be increased at run-time. It is the programmer's task to write application-specific `get_state` and `set_state` methods. An example of how to write such methods will be provided in Section 5.2.4. These methods can also be used to checkpoint the state of an object to non-volatile storage or to perform object migration.

The `view_change` method of an object is invoked whenever another object joins or leaves the group. The `newView` object contains information on the new cardinality of the group as well as the object-references of the group-members.

In Electra, object-group members need to be of the same type, i.e., instances of the same interface, or they must at least have one ancestor interface in common. In the latter case, only the operations inherited from a common ancestor can be multicast to the group.

4.2 Enhanced Environment Interface

When clients invoke an operation on a remote object, a CORBA `Environment` object can be passed as additional parameter. In CORBA, `Environment` objects are used mainly to pass exceptions from the server to the client. Electra enhances the standard `Environment` class with three special operations. By the `call_type` method below, clients specify whether an invocation will be performed synchronously (blocking), asynchronously (non-blocking) or by an intermediary form using a "promise" abstraction [10] to synchronize with the invocation at a later point.

```
// C++
class Environment {
public:
    // Standard Environment interface.
    // See OMG doc. 94-9-14:
    //
    void exception(Exception*);
    Exception *exception() const;
    void clear();

    static Environment_ptr _duplicate(Environment_ptr);
    static Environment_ptr _nil();

    // Electra-specific operations:
    //
    typedef enum {SyncCall, AsyncCall, DeferCall} tCall;
    void call_type(tCall);
    void num_replies(Long);
};
```

The `num_replies` method permits to specify how many member-replies the client's ORB will collect during an invocation. The constant `ALL` demands that the replies of all operational group-members be collected, whereas `MAJORITY` means that the call is active only until a majority of the members have replied. If the constant `COMPARE` is passed to `num_replies`, the ORB collects a reply from each operational member. The replies are then compared and the most frequent one is chosen. If some replies disagree, an exception is raised. If the selection is equivocal on differing replies, another exception is raised. In analogy to comparator mechanisms in fault-tolerant hardware [22], this "poor man's" approach permits to detect faulty system components and to act accordingly. Finally, an arbitrary integral number ranging from one to the cardinality of the group can be specified to collect an exact number of replies. If a majority or a certain exact number of replies cannot be collected due to a failure, an exception is thrown.

5 IDL-Mapping of Group Operations

5.1 Operation Signatures

Electra's C++ mapping goes beyond the OMG specification in that IDL operations are mapped on an additional signature which is needed for non-transparent group communication.

```
// IDL
interface example {
    void op1(in float i, out float o);
    float op2(in float i, out float o);
    void op3(in float i);
};
```

The mapping of CORBA interfaces onto Electra invocation-stubs can be summarized as follows:

- Every operation is mapped into two C++ signatures: one for performing unicast and *transparent* multicast, another one for performing *non-transparent* multicast. In the non-transparent case, environment, return, `out`, and `inout` arguments are mapped into CORBA sequences of the arguments' data types. Using the non-transparent invocation form, programmers gain access to environment objects, return values, `out` and `inout` arguments generated by the individual object-group members. Transparent group invocations, on the other hand, fill the first arriving reply into the arguments and convey the illusion of communicating with a non-replicated, highly available object. Note that a singleton object can be treated like a group with only one member.

- Operations with `void` return-type, for instance `op1` and `op3`, are mapped into Electra operations which can be issued synchronously, asynchronously, and deferred-synchronously. The programmer selects a call type through an `Environment` object acting as additional parameter for the invocation.

- An upcall method can be specified for asynchronous and deferred-synchronous operations. The upcall is automatically invoked with an own thread of execution when the reply to an operation has arrived at the client. The signature of the upcall is determined by omitting all `in` parameters from the signature of the associated interface operation.

- Operations with non-void return-type, `op2` for instance, are mapped into Electra operations which can be issued only synchronously.

By following these rules, the Electra stub generator translates `interface example` into the following C++ class definition. This class serves as static invocation interface to objects of type `example`:

```
// C++
class example: public Object {
    // typedefs for upcalls:
    //
    typedef void (*op1_upcall)(Float o,
        const Environment&);
    typedef void (*op1_upcall_mc)(const FloatSeq& o,
        const EnvironmentSeq&);
    typedef void (*op3_upcall)(const Environment&);
    typedef void (*op3_upcall_mc)(
        const EnvironmentSeq&);

    // signatures for unicast and transparent multicast:
    //
    void op1(Float i, Float& o, Environment&,
        op1_upcall =0);
    float op2(Float i, Float& o, Environment&);
    void op3(Float i, Environment&, op3_upcall =0);

    // signatures for non-transparent multicast:
    //
    void op1(Float i, FloatSeq*& o,
        EnvironmentSeq&, op1_upcall_mc = 0);
    FloatSeq op2(Float i, FloatSeq*& o,
        EnvironmentSeq&);
    void op3(Float i, EnvironmentSeq&,
        op3_upcall_mc =0);
    ...
};
```

In the above example, the first version of the operations permit transparent communication with singleton objects and object-groups. Issued on an object-group, per default the first arriving member reply is assigned to the `out` arguments, `inout` arguments, to the environment, and to the operation's return value. Replies arriving later will be discarded. The second version of each operation has a sequence type in place of its return arguments, and serves for non-transparent multicast. `op2` can be issued only synchronously as it is a non-void operation.

Since the non-transparent form employs sequences for the `inout` arguments, the question arises how an `inout` parameter is passed from the client to the server. Our solution is to store the parameter in the first position of the sequence. In contrast, `out` parameters are unproblematic, since data is passed from the server to the client only.

Note that the multicast version of an operation requires a sequence of `Environment` objects because such objects contain information on exceptions, and

each group-member can report an exception by itself. Analogously to inout arguments, the first element of an Environment sequence informs the run-time of the operation type and of the requested number of replies.

5.2 Examples

5.2.1 Object-Groups and Multicast

In the following code fragment we demonstrate how two object-implementations, impl1 and impl2, are inserted into an object-group. For the sake of simplicity we create both implementations in the same process. For fault-tolerance, impl1 and impl2 would be instantiated on two different machines. After having created impl1 and impl2 in the server process, the BOA::create_group operation is issued to create an empty object-group and to bind the object-reference group to it. Subsequently, impl1 and impl2 are inserted into the group, and the group-reference is registered with a system-wide name server.

```
// C++
// Server site:
//
// Create two implementations of interface example:
_im_example impl1, impl2;
// declare a group-reference:
example_var group;
// Create an object group, insert impl1 and impl2:
BOA::create_group(group);
impl1.join(group);
impl2.join(group);
// Register the group reference with the name server:
NameServer.bind("my object-group", group);
```

```
// Client site:
// Bind to the group and multicast op3:
//
Environment env;
// Obtain the group reference from the name server:
example_var ref =NameServer.resolve("my object-group");
// transparent, blocking multicast (default):
ref->op3(7.3, env);
```

On the client site, an object-reference is bound to the group and operations issued through the reference will be delivered to both impl1 and impl2. Object-implementations may join or leave the system dynamically and operations can be issued as long as there exists at least one operational group-member.

5.2.2 Synchronous and Asynchronous Communication

The next example demonstrates synchronous, asynchronous, and deferred-synchronous object invocation. Note that the example works independently of whether ref is bound to a singleton object or to a group. In the latter case, transparent group-communication is performed.

```
// C++

// upcall procedure for asynchronous invocation:
void op1_upcall(Float outF, const Environment& env){
    // outF holds the result of the asynchronous
    // invocation below.
}

void proc1(example_var& ref){
    Float outF;
    Environment sync, async, defer;
    sync.call_type(SyncCall);
    async.call_type(AsyncCall);
    defer.call_type(DeferCall);

    // Synchronous (blocking) invocation:
    ref->op1(7.3, outF, sync);
    // at this point outF holds the result.
    ...
    // Asynchronous (non-blocking) invocation:
    ref->op1(7.3, outF, async, op1_upcall);
    // outF is undefined. The result will be passed
    // to the upcall.
    ...
    // Deferred-synchronous invocation:
    ref->op1(7.3, outF, defer);
    // outF is undefined.
    // perform local computations ...
    defer.wait(); // suspends the caller only if necessary.
    // at this point outF holds the result.
}
```

The synchronous call suspends the issuing thread until the reply has arrived. After the call, outF contains the result returned by the server. In case that ref is bound to an object-group, the call is suspended only until the first member-reply has arrived, unless this default behavior is changed by the Environment::num_replies method.

In asynchronous mode, the issuing thread is not suspended and outF remains undefined. As soon as the reply is received by the caller's ORB, the op1_upcall method is started with its own thread of execution, and with outF as parameter. If the server has returned an exception it will be assigned to the Environment parameter of op1_upcall.

The deferred-synchronous call works like the asynchronous one, however, by issuing the wait method on the Environment object, the caller is suspended until a reply is received from the server. When wait returns, the outF argument is defined. Thus, the Environment parameter acts like a "promise" object [10] in that it permits the caller to synchronize with the invocation at a later point, and thus to overlap communication with computation.

5.2.3 Non-Transparent Group Invocation

The next example shows the difference between transparent and non-transparent multicast. By using CORBA sequences in place of an operation's return arguments, programmers gain access to the replies of the individual group-members. Non-transparent multicast is useful when group-members perform different tasks.

```
// C++
void proc2(example_var& ref){
    Float outF; FloatSeq outFSeq;
    Environment sync; EnvironmentSeq defer;
    sync.call_type(SyncCall);
    defer[0].call_type(DeferCall);
    defer[0].num_replies(MAJORITY);

    // transparent multicast (as in proc1):
    ref->op1(7.3, outF, sync);
    ...

    // non-transparent, deferred-synchronous multicast:
    ref->op1(7.3, outFSeq, defer);
    // local computations...
    defer[0].wait();
    // outFSeq now contains the replies of
    // a majority of the members:

    for(ULong i =0; i < outFSeq.length(); i++){
        // do something with outFSeq[i]
    }
}
```

5.2.4 Replication, Migration, State Transfer

The next example addresses the implementation of a fault-tolerant domain name server (DNS) whose replication degree can be dynamically varied. Moreover, the DNS can be migrated from one machine to another while clients are constantly using the service.

```
// IDL
interface DNS {
    void install(in string host, in string address)
        raises (ENTRY_EXISTS);
    void host_to_addr(in string host, out string address)
        raises (NO_SUCH_HOST);
    void addr_to_host(in string address, out string host)
        raises (NO_SUCH_ADDRESS);
    void remove(in string host, in string address)
        raises (NO_SUCH_HOST, NO_SUCH_ADDRESS);
};
```

Fed with the above interface declaration, the Electra IDL compiler generates a set of C++ files containing the static invocation interface of the DNS, the server stubs, as well as a file containing a skeleton of the service with one C++ method per operation declared in the interface. This file also provides a skeleton for the state transfer methods of the DNS object which can be completed by the programmer as follows:

```
// C++
void _im_DNS::get_state(AnySeq& state, Boolean& done,
    Environment_ptr){
    // Pack all DNS entries into the "state" object:
    //
    for(ULong i =0; i < 2 * hosts.length(); i += 2){
        state[i] <<= hosts[i/2];
        state[i+1] <<= addresses[i/2];
    };
    // The whole state was read:
    done = TRUE;
};

void _im_DNS::set_state(const AnySeq& state,
    Boolean done, Environment_ptr){
    // Unpack the received DNS entries.
    // First we must clear "hosts" and "addresses":
    //
    hosts.length(0); addresses.length(0);
    for(ULong i =0; i < state.length(); i += 2){
        state[i] >>= hosts[i/2];
        state[i+1] >>= addresses[i/2];
    };
    // We are not interested in "done" since
    // the whole state was transmitted at once.
};
```

To increase the replication degree of a DNS service, a DNS object implementation is created and joined to the respective DNS object group. Electra will invoke the `get_state` object of a group member, marshal the `state` object, transfer it to the newcomer, unmarshal it and invoke the newcomer's `set_state` method. Owing to totally ordered multicast and to Virtual Synchrony, the internal states of the group-members remain consistent in spite of DNS objects joining and leaving the group, and in spite of client applications creating and removing entries from the service while membership changes are occurring.

In order to migrate a DNS object which is a member of a certain group, one just has to instantiate a new DNS object on the destination machine and to join the object to the group. Electra will automatically transfer the state of the obsolete object to the newcomer object and the two DNS objects will run synchronized. Now, the obsolete object can simply be destroyed.

6 Configurability

Electra can run on various toolkits and operating systems, the current version supports Horus, Isis, and MUTS [24]. We believe that Electra can be ported to Amoeba, Chorus, Consul, Transis, and to other platforms providing multicast and threads, without

much effort. Although Electra could be configured to run directly atop of contemporary operating systems such as UNIX or WINDOWS NT, we prefer a solution where a platform like Horus or Isis is employed, since we regard reliable group-communication and Virtual Synchrony as a fundamental part of an ORB.

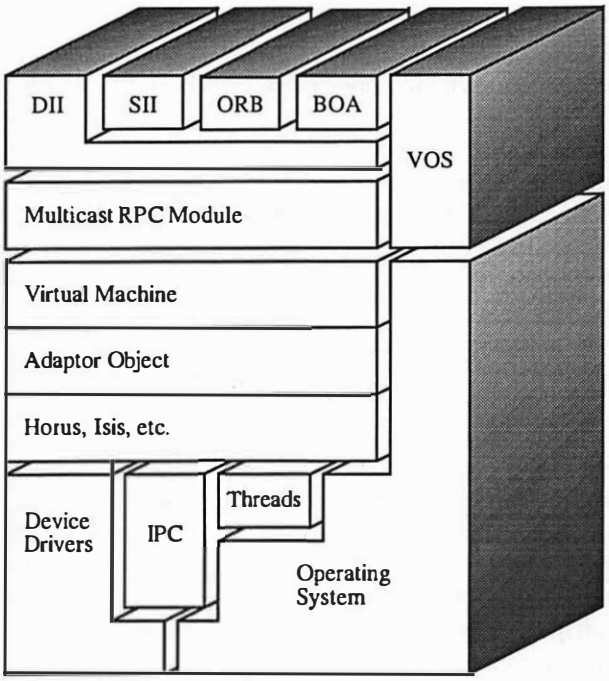


Figure 2: Electra Architecture.

Electra is layered as depicted in Figure 2. The CORBA Static Invocation Interface (SII), Object Request Broker Interface (ORB), and Basic Object Adapter (BOA) are based on the Dynamic Invocation Interface (DII) which can be seen as the core of the ORB. The core itself is built atop of a multicast RPC module supporting asynchronous RPC to both singleton and group destinations. A straightforward approach would consist in building the multicast RPC module directly on Horus, but for the sake of flexibility and portability we decided to base the module on a toolkit-independent veneer, called the *Virtual Machine*.

The Virtual Machine interface exports operations for creating communication endpoints, for aggregating endpoints to groups, for asynchronous message passing, and for lightweight-processes. RPC module and Virtual Machine communicate by means of downcalls und upcalls. The Virtual Machine can be thought of as representing the "least common denominator" of the toolkits to be supported. Determining the "instruction set" of the Virtual Machine was challenging, the resulting Virtual Machine suitable for Horus, Isis, and MUTS is described in [13]. VOS is a Virtual Operating System layer used by applications to interact with the underlying operating system in a portable and thread-safe way.

To map the Virtual Machine interface onto the proprietary API provided by the underlying toolkit, a toolkit-dependent *Adaptor Object* is implemented. An Adaptor Object cleanly encapsulates all of the program code which is specific to a toolkit and necessary to support the multicast RPC module. We call this system-design principle the *Adaptor Model* [11]. To port Electra to a new toolkit, programmers only have to develop an appropriate Adaptor Object. Our adaptors for Horus, Isis, and MUTS comprise less than 1000 lines of C++ code each.

Electra-applications can be reconfigured to run on another toolkit by simply relinking them with the appropriate Adaptor Object. Recompilation of applications is not necessary, therefore applications delivered in binary form can be reconfigured as well.

7 Discussion

7.1 The Direct Approach Would Not Work

One might ask where Electra will give better results than an ORB built on conventional RPC or message passing mechanisms. In fact, conventional CORBA ORBs also provide multicast object invocation, namely through the `send_multiple_requests` dynamic invocation interface operation. This approach works at best when only one client multicasts to an object-group, or with multiple clients and static group membership. However, if two or more clients issue operations on a group, and given that some of the operations alter the state of the objects, the internal states will eventually become inconsistent since multicasts sent by different clients might arrive in different order at the members (Figure 3). Furthermore, if objects need to join and leave the group dynamically, a group membership protocol is necessary to maintain the view[3] that each of the client applications has on the current group membership consistent and for enabling state transfer to newcomer objects. Also, the operations multicast by the clients must be synchronized with view changes. Such ordering and membership protocols are not part of conventional ORB technology but are the basis of Horus and Isis.

Another problem can occur when a causal dependency [9] exists in a chain of asynchronous object

[3] i.e., the array of CORBA `Request` objects passed to `send_multiple_requests`.

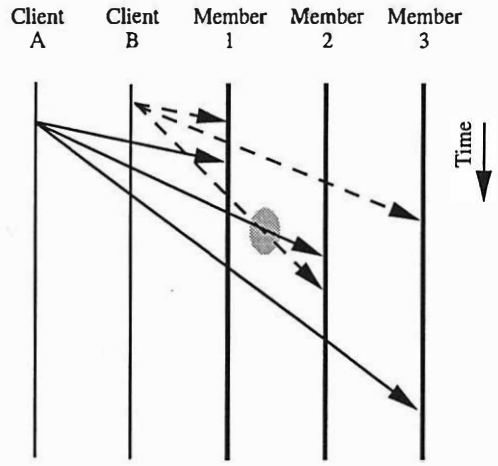


Figure 3: Group operations may be dispatched in inconsistent order if the multicast protocol does not provide total ordering.

invocations. In Figure 4, a client invokes an operation on Object 1 to deposit money on a bank account. Later, the client notifies Object 2 of the deposit. Now, Object 2 tries to withdraw what it believes is on the account, but the deposit operation is delayed due to an overloaded network. Thus, the account is mistakenly overdrawn. Horus and Isis avoid such problems by maintaining causal ordering of messages across multiple processes. Owing to context information appended to messages, the run time system of Object 1 would recognize that a causally preceding message is missing and would delay the withdraw operation until the deposit operation could be dispatched. Conventional ORBs do not guarantee causal delivery.

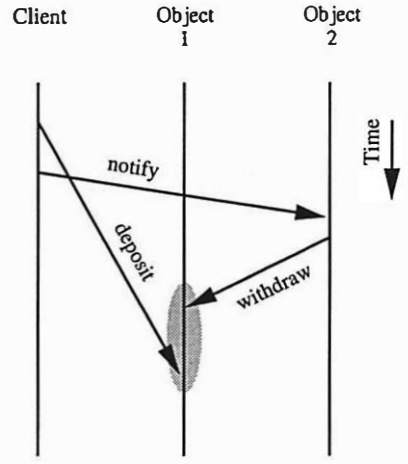


Figure 4: Causality may be violated if the communication subsystem does not guarantee causal delivery.

A further weakness of ORBs that straightly follow the CORBA specification is that oneway operations have only best-effort semantics. This means that a oneway operation can get lost even when no failure occurs, although reliable, asynchronous communication is often necessary for building efficient distributed applications. In contrast, Electra provides reliable, asynchronous communication, and reliable, order-preserving multicast both through the static and dynamic invocation interface.

Last but not least, client applications built on conventional ORB technology may have incongruous opinions on which objects in the system have failed, which can lead to unpredictable behavior of applications. In contrast, systems like Horus and Isis provide failure detection and consistent propagation of failure beliefs to solve this problem.

7.2 The Best of Both Worlds

The thesis underlying our work is that an ORB based on group communication primitives and on the Virtual Synchrony model will considerably simplify the development of robust, scalable distributed systems, and that communication primitives provided by operating systems in widespread use today are not the adequate foundation of an ORB. We regard platforms like Horus, Isis, Transis, and Consul as providing the necessary system support for an ORB, including for instance, fault detection, reliable group communication, and consistent ordering of events. Coming from this position, we described the design and implementation of Electra — a novel CORBA Object Request Broker combining the benefits of OMG CORBA with the strengths of systems like Isis. Electra aims to support applications that require high availability, efficient diffusion of data to large groups of recipients, parallelism, or a combination thereof. Prospective application areas of Electra are video-on-demand servers, video conferencing systems, groupware, distributed parallel computing, and different kinds of fault-tolerant client-server applications.

In addition to the functionality provided by conventional ORBs, Electra permits to aggregate object-implementations to logical groups and to name them as a single unit. Per default, such object-groups are transparent to the programmer and appear like highly available singleton objects. An operation on an object-group will succeed as long as at least one member survives the operation, and the replication degree can be varied at run-time. If required, programmers can break this transparency and gain access to the results generated by the individual members of a group.

Object-groups can be employed for fault-tolerance, load sharing, caching, efficient data distribution, network management, and object-migration. To hide communication latencies, Electra operations can be performed synchronously, asynchronously, or deferred-synchronously, both through the static and dynamic invocation interface. We also have shown how group communication and Virtual Synchrony can be added to a CORBA ORB, namely by enhancing the `BOA` and `Environment` interfaces with a few easy-to-use methods, and by employing Horus or Isis as communication subsystem.

Acknowledgements

The author would like to thank Ken Birman, Roy Friedman, Sophia Georgiakaki, Sean Landis, Robbert van Renesse, Scott Swarts, and Alexey Vaysburd for their support and for their suggestions.

Availability

Please contact `maffeis@acm.org` if you are interested in using Electra. Presently, Electra is a working prototype consisting of a CORBA IDL compiler, a Dynamic Invocation Interface, an ORB Interface, a Basic Object Adapter, and Adaptor Objects for Horus, Isis, and MUTS.

References

[1] AMIR, Y., DOLEV, D., KRAMER, S., AND MALKI, D. Transis: A Communication Sub-System for High Availability. In *22nd International Symposium on Fault-Tolerant Computing* (July 1992), IEEE.

[2] BAKER, S. Multicasting for Sound and Video. *Unix Review* (Feb. 1994).

[3] BIRMAN, K. P., AND VAN RENESSE, R., Eds. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[4] GOLDING, R. A. Weak Consistency Group Communication for Wide-Area Systems. In *Proceedings of the 2nd IEEE Workshop on the Management of Replicated Data* (Nov. 1992), IEEE.

[5] HADZILACOS, V., AND TOUEG, S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, S. Mullender, Ed., second ed. Addison Wesley, 1993.

[6] HERBERT, A. Distributing Objects. In *Distributed Open Systems*, F. Brazier and D. Johansen, Eds. IEEE Computer Society Press, 1994.

[7] JAHANIAN, F., FAKHOURI, S., AND RAJKUMAR, R. Processor Group Membership Protocols: Specification, Design and Implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems* (Princeton, New Jersey, Oct. 1993), IEEE.

[8] KAASHOEK, M. F. *Group Communication in Distributed Computer Systems.* PhD thesis, Vrije Universiteit, Amsterdam, 1992.

[9] LAMPORT, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978).

[10] LISKOV, B., AND SHRIRA, L. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *ACM SIGPLAN Notices 23*, 7 (July 1988).

[11] MAFFEIS, S. A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming. In *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming* (1994), Lecture Notes in Computer Science 791, Springer-Verlag.

[12] MAFFEIS, S. System Support for Distributed Computing. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe 1994* (1994), Lecture Notes in Computer Science 797, Springer-Verlag.

[13] MAFFEIS, S. *Run-Time Support for Object-Oriented Distributed Programming.* PhD thesis, University of Zurich, Department of Computer Science, 1995.

[14] MEYER, B. *Object Oriented Software Construction.* Prentice Hall, 1988.

[15] MISHRA, S., PETERSON, L. L., AND SCHLICHTING, R. D. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. *Distributed Systems Engineering Journal 1*, 2 (Dec. 1993).

[16] NICOL, J. R., WILKES, C. T., AND MANOLA, F. A. Object Orientation in Heterogeneous Distributed Computing Systems. *IEEE Computer 26*, 6 (June 1993).

[17] OBJECT MANAGEMENT GROUP. *IDL C++ Language Mapping Specification*, 1994. OMG Document 94-9-14.

[18] OBJECT MANAGEMENT GROUP. *The Common Object Request Broker: Architecture and Specification*, 1995. Revision 2.0.

[19] PETERSON, L., BUCHHOLZ, N., AND SCHLICHTING, R. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems 7*, 3 (Aug. 1989).

[20] SCHIPER, A., AND RICCIARDI, A. Virtually-Synchronous Communication Based on Weak Failure Suspectors. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing* (Toulouse, June 1993), IEEE.

[21] SCHLICHTING, R. D., AND SCHNEIDER, F. B. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems 1*, 3 (Aug. 1983).

[22] SIEWIOREK, D. P., AND SWARZ, R. W. *Reliable Computer Systems: Design and Evaluation.* Digital Press, Bedford, Mass., 1992.

[23] SOLEY, R. M. *Object Management Architecture Guide.* Object Management Group. OMG Document 92-11-1.

[24] VAN RENESSE, R. A MUTS Tutorial. MUTS Documentation, Cornell University, 1993.

[25] VAN RENESSE, R., AND BIRMAN, K. P. Fault-Tolerant Programming using Process Groups. In *Distributed Open Systems*, F. Brazier and D. Johansen, Eds. IEEE Computer Society Press, 1994.

[26] VAN RENESSE, R., AND BIRMAN, K. P. Protocol Composition in Horus. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (Ottawa, Ontario Canada, Aug. 1995).

[27] VERÍSSIMO, P., AND RODRIGUES, L. Group Orientation: A Paradigm for Distributed Systems of the Nineties. In *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems* (Apr. 1992), IEEE Computer Society.

Using meta-objects to support optimisation in the Apertos operating system

Jun-ichiro Itoh. *Keio University, Kanagawa, Japan.*
Rodger Lea. and Yasuhiko Yokote. *Sony CSL, Tokyo, Japan.*

Abstract

The Apertos OS has explored the use of the meta-object model and reflection as a means to build highly flexible operating systems. While the benefits of such a system are great, the performance cost of a clean consistent use of the meta object model is high. Our initial work accepted this because of our desire to explore the model to its fullest. Recently we have turned our attention to optimisation techniques. Our work is interesting because we have tried to use the flexibility of the model to support our optimisations. This is in contrast to many systems that restrict or even compromise their initial model for the sake of performance. This paper outlines the Apertos system and discusses the optimisations we have made.

Introduction

Recent trends in networking, multi-media and mobile computing have placed new requirements on the flexibility and adaptability of existing operating systems. It is increasingly recognised that existing OS architectures are not sufficiently flexible to meet these demands. As such there is significant interest in building more flexible, or adaptive operating systems. However, this flexibility often has a performance price and OS designers are faced with a trade off between flexibility and performance. Inevitable, performance becomes the overriding factor, and the clean conceptual model originally proposed to achieve flexibility is compromised.

In this paper we describe our work with the Apertos operating system which is based on the meta-object model. We outline the design and overall implementation of the system and show how we have explored the meta-object model to its fullest. In particular we discuss how we have used the meta-object model consistently throughout the entire system. Our initial implementation suffered from this desire for a clean model by having poor performance. Recently we have been exploring how to optimise the system. Our work differs from much other optimisation exercises in that we are endeavouring to exploit our model to support optimisation, rather than breaking the model. We feel that our initial experience is promising, and that the techniques we are adopting will be beneficial to other groups who are exploring the object oriented paradigm for the construction of operating systems.

This paper begins with an overview of the different approaches to system flexibility; it then briefly describes the object/meta-object model; we present an overview of our implementation of this model in the Apertos system; finally we discuss our current work to optimise our system within the framework of the object/meta-object model.

*itojun@mt.cs.keio.ac.jp

OS flexibility

We can categorise flexibility in work to date as follows: flexibility in the way we combine the components of the system at build time, flexibility in the way the systems adapts to application needs and flexibility in the way the system and its interfaces evolve over time.

Flexibility though modularity: Modularity is a key technique in building flexible software since it encapsulates components and makes them easier to change or evolve. Micro-kernel architectures have popularised such modularity and have demonstrated benefits in their overall flexibility[3][2]. Object oriented operating systems have extended this modularity with a finer degree of granularity and used techniques such as abstract interfaces and inheritance hierarchies to support a high degree of flexibility[10]. Systems such as Choices[6] have exploited this software engineering model to support an extremely sophisticated mix and match approach to the initial build phase, in some cases exploiting it to specialise systems not only for hardware but for the intended applications.

Flexibility through run-time adaptation: Most operating systems have supported some degree of run time adaptation; for example scheduling policies are often adapted to deal with changing application loads. Recently there has been much interest in the use of more sophisticated techniques to not only choose between existing policy modules, but also to dynamically generate code that is optimised to a set of run-time constraints [13] [4].

Flexibility of interface evolution: Flexibility to support the evolution of system interfaces is more difficult to achieve. Some of the work on object oriented systems has attempted to exploit type conformance [5] or similar means [7] to achieve it.

It is clear that the operating system community is aware of and is actively addressing the issues of flexibility. We argue however, that although this work is moving in the right direction, it is ad hoc in that is opens up pieces of a system without an overall architectural model for making that flexibility intrinsic and so available to all OS components.

Reflection and its use by Apertos

The Apertos system (formally Muse)[15][16] has been investigating a flexible open OS architecture since 1988 and has concentrated its efforts on the meta-level programming model.

Simply put, a meta-level architecture is one in which all system components have a meta-level component that holds information about that component and how it is used; in essence the meta-level defines control and policy. This meta-level component can be viewed as an environment for the component and provides an interface to manipulate aspects of the component's environment. This is traditionally called the meta-interface or meta-object protocol (MOP)[9]. However it could equally be called its management interface.

For example, a thread in an object oriented operating system is represented by a simple object that has a 'standard' interface such as start and yield. In addition, somewhere in the kernel is the scheduling code which decides which thread to execute next. This algorithm is often referred to as the policy associated with the simple object. In our terms this algorithm is part of its meta-object. In a traditional OS the implementation of the policy and its application to the thread was a closed a-priori decision made by the OS designers. In a more flexible system we would like to be able to open up that algorithm, make it accessible to other parts of the system and perhaps even change it. By providing an interface to the scheduler meta-object, a MOP, we are able to expose that algorithm and allow system designers and even applications to use it.

Reflection can be viewed as a use of this

basic foundation. Reflection is the process in which we step back from computation that is on behalf of, or part of the application and carry out computation on behalf of the system itself. To achieve this it is necessary to be able to access a description of the system. When we have finished this computation we will have changed the description of the system. These changes are then 'reflected' back to the actual system. Hence reflection gives us a way to cleanly make changes to the system. Using the example above, when we use the MOP interface to make a scheduling decision, e.g. reorder the scheduler queue, we are making a reflective computation at the meta-level.

It is clear that this ability to access and manipulate the basic features of an OS is useful; some of the work cited above is using a variety of different techniques to do just that. What is different about the Apertos system is that the entire operating system is built using this meta-object architecture and so all aspects of the system are open and can be easily changed. It is our belief that only by doing this in a completely consistent manner will we be able to fully realise the benefits of the flexibility it provides.

The Apertos operating system

As mentioned above, Apertos is an investigation into how we can apply this meta-object model to all aspects of the operating system and what are the costs and drawbacks. The Apertos system generalises some of the ideas mentioned above.

Each simple object in Apertos is managed by a set of meta-objects. The meta-objects are associated with the simple object via a descriptor object that holds a reference to all the required meta-objects. This set is referred to as a meta-space which is also represented by an object (which we call a reflector). The meta-space constitutes the execution environment for an object. At any point in time there will be many meta-spaces co-existing in the system.

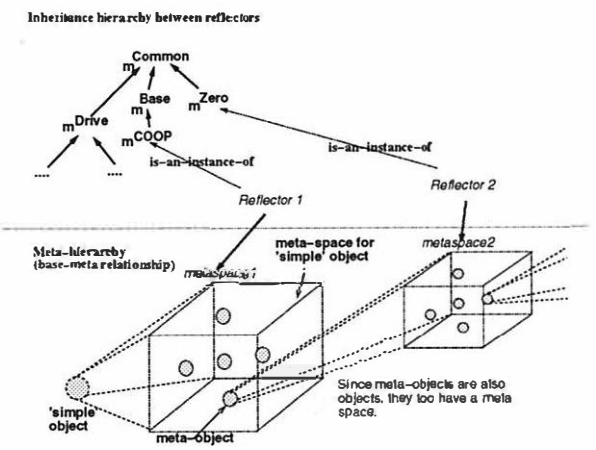


Figure 1: The object/meta-object relationship and the reflector hierarchy

Interaction between objects is achieved through MetaCore, a minimal system component that allows an object to call into its meta-space. MetaCore can be likened to a micro kernel, however rather than implementing a simple set of base abstractions, it implements only the means to move from a simple object into its meta-space.

An object that wishes to access another object does so by calling into its meta-space; the target object is located and the invocation dispatched. In the cases where an object wishes to compute not at the base level, but at the meta-level, the same invocation model is used. However the target for the invocation is a meta-object and not a base object. This common invocation path implies that all aspects of the invocation are also open to change including look-up, dispatch and invocation semantics.

The last extension that Apertos supports is a means of allowing objects to change their behaviour not by using the MOP to modify some feature of their existing environment, but by migrating themselves to a completely new environment which supports a different set of policies. Apertos supports a simple form of type conformance to allow this.

Implementation

In the previous sections we outlined the conceptual framework that Apertos supports, i.e. the object/meta-object model, and discussed how the Apertos system conforms to this conceptual framework. In this section we discuss in more detail how the actual implementation of these concepts is carried out.

In fig.2 the abstract meta-architecture shown in fig.1 is visible as the separate meta-spaces *m*COOP, *m*Zero and *m*Core.

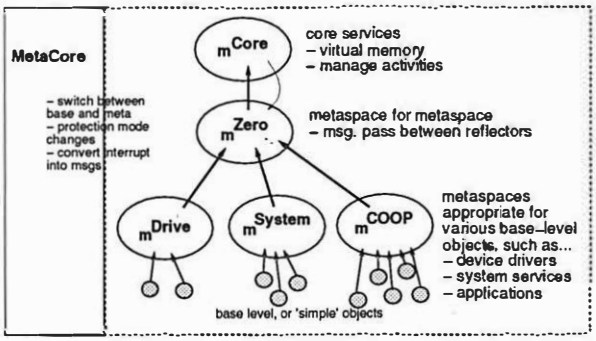


Figure 2: Implementation hierarchy of meta-spaces

MetaCore

MetaCore acts in a manner similar to that of a micro-kernel in more traditional systems. However, rather than providing a minimal set of resource abstractions, its role is restricted to supporting the object/meta-object model, the base notion of activity and interrupt dispatching. This allows a very small core object (currently around 3K) which aids both understanding and porting. MetaCore supports the following operations:

- M(meta): Make a request for meta-computing. This causes the execution of an object to be suspended and control to pass from the object to its meta-object.
- R(reflect): Resume object execution, i.e. pass control from the meta-object back to a base object.
- CBind: Specify a recipient for an interrupt. Interrupts will be delivered to the recipient as messages.
- CUnbind: Break the association between a recipient and an interrupt.

MetaCore is shown in fig.2 as the vertical box running alongside the meta hierarchy. Since MetaCore supports the basic facilities for meta-computing, i.e. the reify(M) and deify(R) operations, it is accessible and used by all meta-spaces. In table 1 we give performance figures for these basic operations[1]. Note that operations without trap occur when we do not cross a protection boundary.

Table 1: Execution cost of *MetaCore* primitives (in μsec)

primitive	*on i486*
M	21.1
M(w/o trap)	13.0
R	22.6
R(w/o trap)	8.8
CBind	4.1
CUnbind	3.5

The key meta-spaces

***m*COOP** is the first level meta-space for a base level object, it implements the scheduling policy and provides a concurrent object programming model for base objects.

***m*Zero** is one of the terminal meta-spaces for all other meta-spaces and is thus accessed by all other meta-spaces. This allows some operations, eg. context location. to be optimised.

[1] All performance figures from a 486DX2-66MHz, 16MByte PCAT compatible.

m^{Core} is the reflector for m^{Zero} and contains meta-objects to handle activity, physical memory and virtual memory. Although, as mentioned above, m^{Zero} is the shared reflectors for all others, it too needs a reflector, hence the use of m^{Core}. To halt to potentially infinite hierarchy of meta-spaces we make the meta-space for m^{Core} be m^{Zero}. Although this introduces a circular dependency between the two it allows us to 'bottom out' when moving down through the meta hierarchy.

Activity

The underlying CPU is abstracted in a structure known as a **context**. Contexts[2] are created by the meta-object *exec* which resides in m^{Core}. Contexts are actually abstracted by a notion of **Activity** which is the unit of manipulation at the meta-level (see fig.3). The context structure is seen by MetaCore as it uses this to set up the hardware to support the actual execution of an activity. The context structure contains the pointer to the meta-space for that context. This enables MetaCore to move from the currently executing context to its meta-space when the M operation is called.

Optimisations

Our initial work on optimisation has concentrated on its use in the lowest levels of the OS, i.e. the hardware interface. We begin this discussion by outlining some of the details of the driver metaspaces.

m^{Drive} and m^{System} are meta-spaces for device driver programming (see fig.2). They implement a concurrent object environment where device drivers are written in the same way as any other concurrent object application, i.e. we are able to hide the restrictions normally visible to device

[2]Contexts in Apertos should not be confused which the general OS concept of a context as an address space. They are simply the state of the machine registers

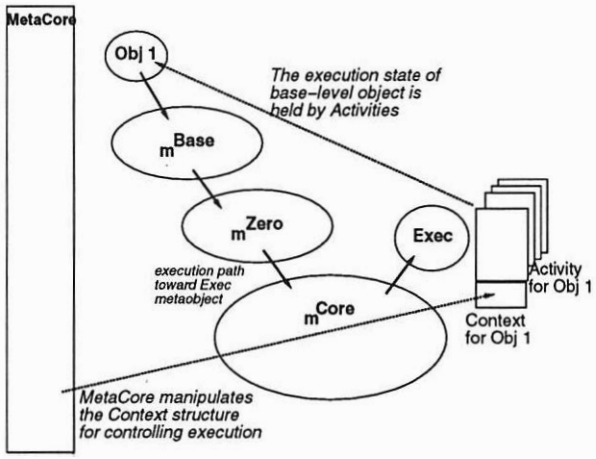


Figure 3: Activity and the execution path to exec metaobject

driver programmers. m^{Drive} supports device driver objects that have direct physical access to hardware and manage interrupts, and m^{System} supports device driver objects that have no access to hardware.

We implement device drivers as single-threaded concurrent objects. The advantage of this is that we can completely hide any complex mutual exclusion operation normally associated with device driver programming. Instead the details are managed automatically by the meta-objects in the device driver meta-space. Device driver objects are programmed without recourse to shared data structure and communication between objects is carried out by passing messages. Service entry points and interrupt handlers are implemented as methods of device driver objects which are invoked on message reception. In this way we can provide a safe programming environment for programmers who write very low-level system programs.

One complication when writing device driver objects is that we need to have a way to synchronize interrupt handler methods and hardware manipulation methods so that we can cleanly switch between hardware manipulation and high priority interrupts. We achieve this in an object-oriented

fashion by using continuation meta-objects. Continuations perform a similar role to 'futures' as used in languages such as Actor[1] and ConcurrentSmalltalk[14]. However we use them for synchronizing methods of an object, not to synchronize between two objects.

Table 2 gives some brief details of execution costs for interrupt handling and using concurrent objects in the driver meta-spaces. Details can be found in [8].

As can be seen our approach has been to maintain the concurrent object oriented model supported throughout Apertos even at the device driver level. We believe that this approach provides a cleaner way to write device drivers, which has always been something of a black art, and hides some of the inherent complexity within the meta-space mechanisms.

Context hand-off

Because our system uses the object/meta-object model throughout, the general case execution path constantly moves from base to meta. As discussed above, we are using a concurrent object model so that each object (including meta-objects) has it's own execution context. Thus our system is forced to constantly switch contexts as it traverses the meta-space hierarchy. Obviously our first target for improvement is in the implementation of this context switching. We have implemented a variant of thread hand-off which we refer to as context hand-off.

Context hand-off will occur when an object issues a synchronous (Call-and-Reply style) method call to another object. In our basic implementation, there will be at least 4 context switches[3] when issuing synchronous method call, i.e. switch to reflector for call, go up to callee, switch to reflector for reply, go back to caller. If it is safe to share the execution context the context hand-off mechanism will use a simple function call for the synchronous method call. This means that the caller's context will be used for executing the callee's method.

The following conditions should be satisfied to use context hand-off mechanism:

- caller and callee share the same address space, and are in the same protection boundary.
- various state information, such as interrupt masks or CPU execution mode, kept in the execution context are the same.
- it is acceptable to skip the scheduler when calling into the reflector. If the timeslice for the caller has expired, or will soon expire, then skipping the scheduler is not acceptable.

It is the job of the reflector to decide if we are to use the context hand-off mechanism or not. It manages various state values on behalf of base-level objects and monitors behaviors (every external event from an object will go through reflector).

This use of state information in the reflector is a classic case of the use of meta-data as a basis for a reflective operation. The reflector uses the data that describes the state of the base object to make a decision about how a system operation, in this case synchronous call/reply is implemented. When it decides that a context hand-off is acceptable, it manipulates the implementation of the mechanism to optimise the code path. I.e., it carries out computation at the meta-level and reflects the changes back into the system.

To ensure sufficient application control, a meta-level method which allows context hand-off to be explicitly disabled by a request from the base-level objects is available if needed.

The performance benefits of this technique come from the avoidance of context-switches and are bought at the small cost

[3]It is important to bear in mind that we use the term context switch here to mean a lightweight switch between two concurrent objects, not as a heavier weight notion consiting of a complete address space switch.

Table 2: Costs of mDrive services, and interrupt operations (in μsec)

metaoperation	*Apertos*	*BSD/386*
Interrupt message delivery	25.0	11.5
Null interrupt handler execution	44.2	16.2
Send metacall overhead on mDrive	108.6	—
Call-Reply roundtrip on mDrive	207.8	—

of the decision making code held in the reflector. The performance benefits from this simple technique are impressive, in table 3 we show the benefits gained within the driver environment with the use of context hand-offs. In this evaluation, context hand-off avoids fourteen context switches since we normally need context switches between objects and between schedulers which are also implemented as single-threaded concurrent objects with reflectors. It should also be noted that in the case of the context hand-off decision failing, i.e. deciding that the four criteria mentioned above are not met, then the performance is not affected.

To confirm that these benefits are consistent at a macroscopic level we have used the context hand-off mechanism in the implementation of the IP protocol handlers. Based on our design principle, the network protocol handlers are implemented onto an appropriate system service layer for protocol handlers [12]. We have implemented IP network protocol handler objects onto the system service layer mSystem. There is a concurrent object per each network protocol, five in total, passing IP packets as message between them. We have measured the execution time after reception of ICMP echo packet through the transmission of ICMP echoreply packet. Table 4 shows the results, indicating that optimised execution is 10 times faster than the non-optimised execution[4] In the normal case there will be at least 54 context switches (dependent on timing conditions such as timer interrupts) during the execution of the non-optimised execution. In the optimized execution we avoid all the context switches.

[4]We do not provide performance figure for the UNIX variant because the implementations are completely different.

Message batching

We have also implemented a technique which exploits message asynchrony to allow us to batch messages based on message send patterns.

Again we use the meta-object model by using reflectors to gather statistics on message communication patterns. As discussed above, this is possible because all message communication goes through the reflector. We then allow the meta-space to re-configure itself to change the semantics of the send from a simple send per invocation to a batched send model. As before, the meta-space can be configured to not batch asynchronous messages by invoking a meta-level method from the base-level object.

Fig.4 gives a graph of the initial performace speed up using this batched message send technique. The horizontal axis indicates the number of asynchronous messages to be actually batched into one packet; the vertical axis indicates the time to process all the message sending requests. As you can see, if the batching is enabled and there is only one message in a packet, there is a slight overhead. The source of the overhead is the processing time for batching. We can reduce execution time if we can batch 2 or more messages into single packet.

While the use of message batching is in itself not a new technique, the ease with

Table 3: Costs of m^{System} services (in μsec)

metaoperation	*Apertos*
Call-Reply roundtrip on m^{System}(context hand-off not implemented)	772.8
Call-Reply roundtrip on m^{System}(context hand-off succeed)	7.8
Call-Reply roundtrip on m^{System}(context hand-off failed)	693.0

Table 4: Costs of network protocol handlers on m^{System} (in μsec)

metaoperation	*Apertos*
ICMP echo-echoreply roundtrip	3466.0
ICMP echo-echoreply roundtrip(optimised)	362.3

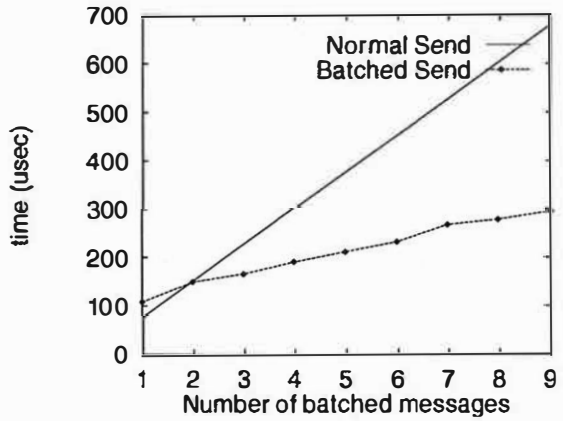


Figure 4: Effect of simple message batching

which we can add this technique to our system is a function of the meta-space architecture. Since the system implements message send as a meta-operation, and since we have a meta object that describes how that operation is performed it becomes simple to use reflection to change the way we send messages. Again, at the implementation level, because our model moves from base to reflector on all meta-operations, it provides a convinient place to monitor system operation. In this case we simply instrument the reflector to monitor message traffic, that is, make message traffic statistics into meta-data, and use this meta-date as the basis for the reflective operation.

Dynamic object reallocation, and context handoff

A related technique we have adopted is the use of address space change combined with context hand-off. The context hand-off technique only works when two objects share the same protection domain limiting its use. However, Apertos implements every system entity using fine-grained concurrent objects allowing several objects to be allocated onto a single address space. In Apertos, unlike UNIX and other operating system with fixed design policies, various aspects of the system such as protection, activity (CPU thread allocation), object persistence, and message passing semantics can be tailored to meet the needs of the system designers. Therefore, if performance is the critical issue for the objects and protection is not an issue, (for example they are written using a language system with strong type inference) the system designer can use a meta-space which allocates objects onto the same address space and always utilizes the above described optimisation techniques. If per-

formance is not an issue and protection is important, one can provide a meta-space which implements strict protection policies. An object can choose one of these policies on the fly, by migrating between two meta-spaces.

Again Apertos utilizes fine-grained concurrent object, these objects can be moved from the current address space to other address spaces. By doing so, we can tailor object allocation on a per address space basis allowing us to maximize the utilization rate of the optimisation techniques. This technique resembles the load balancing techniques used between remote CPUs.

Run-time code generation

The last area of work that we have explored is the use of run time code generation and partial evaluation. It is often the case that we can optimise the general case execution path provided at initial system load based on application requirements.

Again we use the meta-space to gather run time information about the execution characteristics of the application. By using the meta-object interface we are able to adapt the code execution to the application requirements.

This work is based on our initial investigation in the Cognac language [11]. Cognac uses static analysis and type information to drive run-time binding of an object to its meta-objects. By dynamically changing these run-time bindings we are able to maintain the same base programming interface, but change the semantics of the system level operations that support the application object. Also we are investigating ways to utilize partial evaluation techniques to flatten out meta-object hierarchies, which can then be statically linked into the base-level object.

Conclusion

We believe that operating system flexibility is the most important goal of current research. However we would argue that flexibility must be provided within the framework of a clean and consistent architecture. Our approach, the Apertos operating system, uses the meta-object model and is targetted toward implementing the whole operating system within a reflective programming model. This approach has produced an operating system which is highly flexible at all levels, however, it has some performance drawbacks. In this paper we have discussed ways to minimize the execution cost by using the meta-object model in full and have shown that it is possible to exploit meta-object to gain significant performance advantages without breaking the model.

Current and future work

We are currently working on the use of Apertos in a distributed multi-media environment. In our initial set up we have ported Apertos to a number of Sony News 5000 workstations with MIPS R4000 processors and a video server using the R3000 processor. These machines are interconnected with a small experimental ATM network. We are experimenting with Apertos in two areas; the first uses the meta-object model to support QofS constraints and exploits our flexible architecture to adapt the resource scheduling policies to varying simulated system loads and failures. The second area of work uses Apertos in a proprietary graphics engine attached to the above network where we are experimenting with protocols for distributed shared virtual environments. In particular we are looking at the issues of distributed consistency where we are exploring the use of migration between meta-spaces to dynamically change the consistency associated with shared objects. Also, we plan to investigate the use of object migration (to the remote host) for downloading the whole system code including device drivers, to the graphics engine. Using meta-spaces as a clear abstraction for various visibility control, such as host boundary, persistent

objects and volatile objects, is also planned and will be carried out in the next development phase.

In addition we are continuing to analyse the performance of our system and hope to extend the techniques discussed above to all meta-spaces. We hope to report in the future on the success of this work.

Availability of Apertos Operating System

The Apertos system is available for research purpose and information, along with a collection of papers can be found by accessing the following URL:

`http://www.csl.sony.co.jp/`
`ftp://ftp.csl.sony.co.jp/`

References

[1] Gul A. Agha. Actors: A Model Of Concurrent Computation In Distributed Systems. Technical Report 844, MIT, June 1985. Technical Report 844.

[2] Tevanian Avadis Jr. and Richard F. Rashid. MACH: A Basis for Future UNIX Development. Technical report, Department of Computer Science, Carnegie Mellon University, June 1987.

[3] Nariman Batlivala, Gleeson Barry, Hamrick Jim, Lurndal Scott, Price Darren, Soddy James, and Abrossimov Vadim. Experience with SVR4 over CHORUS. In *Micro-kernel and other kernel architectures*, pp. 223–241. USENIX, April 1992.

[4] Brian Bershad. SPIN - an extensible micro-kernel for application specific operating system services. In *Proceedings of the 6th ACM-SIGOPS European Workshop: Matching Operating Systems to Application Needs*, September 1994.

[5] C Bryce, V Issarny, G Muller, and I Puaut. Towards safe and efficient customization in distributed systems. In *Proceedings of the 6th ACM-SIGOPS European Workshop: Matching Operating Systems to Application Needs*, September 1994.

[6] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. *Choices*, Frameworks and Refinement. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pp. 9–15. IEEE Computer Society Press, October 1991.

[7] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *USENIX 1993 Summer Technical Conference Proceedings*. USENIX Association, June 1993.

[8] Jun-ichiro Itoh, Yasuhiko Yokote, and Mario Tokoro. SCONE: A New Execution Model for Low-Level System Programming based on Concurrent Objects. Technical Report SCSL-TM-95-005, Sony Computer Science Laboratory Inc., 1995. submitted to OOPSLA95.

[9] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[10] Rodger Lea, Christian Jacquemot, and Eric Pillevesse. COOL: System support for distributed programming. *Communications of the ACM*, Vol. 36, No. 9,, September 1993.

[11] Kenichi Murata, Nigel Horspool, and Yasuhiko Yokote. Design and specification of cognac. Technical Report SCSL-TM-94-006, Sony Computer Science Laboratory Inc., 1994.

[12] Kenichi Murata and Yasuhiko Yokote. A Reflective Network System Using

Concurrent Objects and Meta Architectures. Technical Report SCSL-TM-93-010, Sony Computer Science Laboratory Inc., July 1993.

[13] Carlton Pu and Jon Walpole. A study of dynamic optimisation techniques: Lessons and directions in kernel design. Technical Report CS/E 93-007, Oregon Graduate Institute (OGI), April 1993. Technical Report CS/E 93-007.

[14] Yasuhiko Yokote. *The Design and Implementation of ConcurrentSmalltalk.* World Scientific Publishing, 1990.

[15] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1992.* ACM Press, October 1992. Also appeared in SCSL-TR-92-014 of Sony Computer Science Laboratory Inc.

[16] Yasuhiko Yokote, Gregor Kiczales, and John Lamping. Separation of Concerns and Operating Systems for Highly Heterogeneous Distributed Computing. In *Proceedings of the 6th ACM-SIGOPS European Workshop: Matching Operating Systems to Application Needs*, September 1994.

The Spring Object Model

Sanjay R. Radia, Graham Hamilton, Peter B. Kessler, and Michael L. Powell

SunSoft, Inc.
2550 Garcia Avenue, MTV19-216
Mountain View, CA 94043
Sanjay.Radia@Eng.Sun.Com

Abstract

The Spring Object Model provides a basis for building operating systems, applications, and other software components for a modern distributed computing environment. All services and abstractions— whether local or remote; system, extension, or user; library or server—are structured as objects. Objects have strongly typed interfaces specified in an interface definition language. Spring forces a clear separation of interface and implementation; no implementation properties are allowed in the interface. It supports multiple interface inheritance, which is used to structure the system abstractions and provides the basis of extending and evolving them.

Interfaces are used to define boundaries of software components which can be mapped to different address space and machine boundaries. The model provides parameter passing modes that are especially useful for distributed computing but which can be optimized for the local case. This allows one to construct microkernels with the option of configuring components in the same address space for improved performance.

Objects are instances of interfaces, on which clients can perform operations. A client of an object is generally unaware of the location of the object's implementation, and can perform operations on the object and pass the object around freely. The representation of a Spring object is not a fixed piece of information such as a unique identifier; instead, it can be tailored to meet different needs using the subcontract abstraction.

1 Introduction

The Spring distributed operating system is the result of a research project to develop technology that addresses the problems of building, supporting, and using both system and application software in a distributed computing environment. This paper describes the interface and object technology we have developed and refined over the last five years and our experience with it. The design has been shown to work for a prototype distributed and secure operating system that has a flexible and powerful virtual memory system[13], name service[18][19], and file system[14]. Our technology allows the system to be easily extended and evolved, and writing distributed services and applications in Spring is much easier than in earlier systems. Some of this technology have been transferred into products and industry standards, and that process is ongoing.

Spring has been strongly influenced by ideas in programming languages such as data abstraction, strong typing, explicit interface definitions, complete separation of interface from implementation, inheritance, and, of course, the use of objects. Unlike other approaches, however, we do not assume a single programming language, nor do we assume that the type system is enforced by the operating system. Spring also builds on current good ideas in operating systems such as modular structure, multiple address spaces, transparent distribution via remote procedure calls, multiple programming languages, multi-threading, secure micro-kernels, etc. Unlike most systems, however, Spring uses objects and interfaces technology as the only software-visible structuring mechanism, for all kinds and levels of software.

1.1 Motivation

The Spring project was motivated by the problems the industry faces in building, supporting, and using software in a modern computing environment. These same problems arise when building subsystems and large applications, and our solutions are applicable to those areas as well. Some of the critical objectives are:

- security, including the ability to have different security policies from strict to unchecked,
- uniformity, from single machines to heterogeneous, enterprise-wide networks,
- consistent application model, especially for end-users, but also for administrators and developers, and
- extension, configuration, and evolution, allowing incremental change and coexistence of variations.

The object model plays a key role in handling these issues because it sets the global expectations and policies for how software components interact. In fact, simply having a single, consistent object model can help achieve important properties.

Security

Secure systems have typically been more difficult to use, because security constraints can deny service at unexpected or inconvenient times. Making a system that is truly secure requires the security policy to be rigorously implemented by all components, and almost any lapse can be fatal. We desire a solution in which most software works unchanged in environments with different security requirements.

Uniformity

Most systems use different mechanisms and different programming paradigms to connect components at different granularities. For example, a developer writes different code to access a library, an operating system service, or a network service. It is also common for the equivalent service to use different interfaces, depending on the scope or complexity of the implementation. We want a solution that allows system services to be accessed the same way independent of their implementation.

Consistent Application Model

Users can be confused when presented with a variety of inconsistent and incompatible concepts. For example, in some systems, documents, users, printers, machines, and applications are named, created, and manipulated in different ways primarily because of the different underlying implementations. Even when attempts are made to add a consistent layer on top of disparate mechanisms, problems can arise when the limitations of the implementation cause unexpected constraints that are difficult to explain at the user level. We would like a solution in which user-level abstractions correspond directly to a consistent application model.

Extension, configuration, and evolution

The rapid pace of technology advancement means that new functionality is continually being introduced. As the network grows, more powerful mechanisms are needed to provide the security, performance, robustness, etc., of a smaller environment. Some capabilities that are considered optional for some people may be considered essential for others. We need to treat all of this software in a first-class way, so that it is just as easy to use the old as the new, the extension as the built-in, the special as the general.

Even though many operating systems are implemented using high-level languages, they are hard to change, extend, and evolve. Those systems are not designed as a set of replaceable components. Component interfaces and the interactions between components are not well defined. Changes to one part of the system can easily affect another part. The procedural interfaces for a component are often the outcome of a particular implementation rather than an interface designed to define a replaceable component.

1.2 Tactics

To address these issues, the Spring object model makes a set of decisions, and applies them uniformly to all software in the Spring system. Although higher-level software could in principle obscure those choices, it is our intention for the model to "show through" to programmers and users of the system. We also apply the model down to the lowest level of the system, and use it to design the fundamental mechanisms.

Object-orientation

All software is structured as objects. Instances of abstractions such as files, documents, applications, etc., as well as services such as naming, virtual memory, user interface, etc., are modeled as objects. Functionality is accessed by invoking operations on those objects.

Strong interfaces defined in an interface definition language

An object has a type that is defined by its interface, which specifies the operations that can be performed on the object. Multiple inheritance of interfaces is sup-

ported, allowing objects to be taxonomized by their interfaces. The interface definition language plays a central role in the organization of the system and the implementation of its fundamental mechanisms.

Common programming model

Programmers write the same code to access objects whether they are local or remote, library or server, system, extension, or application. Objects can be accessed the same way independent of their location and implementation technique, allowing address space and machine boundaries to change without changing the software. New or optional services are accessed the same way as existing or built-in services.

Separation of interface and implementation

Multiple implementations of the same interface can coexist and alternatives can be substituted without requiring changes to the users of those objects or other components. Generally, components depend only on the interface to other components, not on the particular implementations.

Capability-style security

Possession of the object confers the right to use the object according to its interface. Objects cannot be arbitrarily constructed, but are acquired via operations on other objects, allowing appropriate access checking to take place. Authority can be delegated by passing objects to other components.

Abstract objects

Unlikely many object systems, Spring objects are abstractions in that their behavior and properties are only perceived through their interfaces. Notions such as the identity, location, or state of an object are meaningful only to the extent that they are exposed through the object's interface.

Type system by convention

Although the object model and type system provide substantial descriptive and organization power for Spring, and they are used pervasively, type information is always considered in light of trust and location. Thus the system cannot be subverted by breaking the type system, and type information can be extended and federated.

1.3 Other approaches

Microkernels have been used to unify mechanisms and address modularity and security issues in systems. Thoth[6], the V system[7], Amoeba[16] and Mach[1] use the microkernel approach. In these systems, software boundaries are tied to hardware boundaries: the system is broken into components, each of which resides in a separate address space, communicating via message passing or remote procedure calls. This approach has a number of limitations. The interface is defined by the request-response message protocol. Replacing a component by another that obeys the same message protocol is fairly straightforward, but extending the interface is more difficult. A well-defined, strongly-typed interface is preferable. Also, the restriction that the components must reside in separate address spaces is a performance penalty. While the firewalls of separate address spaces are often useful during development, it is often desirable in production use to put some components in the same address space and rely on local procedure calls to obtain better performance. Others have reported on other performance penalties of microkernel that use separate address spaces[5].

Language-based operating systems such as Smalltalk and Cedar reduce the gap between the system and the programming environment, and are more modular and easier to change. However, they have restrictions, such as the use of a single language and a single address space, and their type systems must be enforced for correctness. These limitations are unacceptable for an open, secure, heterogeneous, enterprise distributed computing environment.

Other object based systems have focused on different issues and used different tactics. Choices[4][15], an object oriented operating system, has focused on using the object oriented features of the C++ language for building a highly modular kernel for tightly coupled multiprocessors. All system services run in the kernel address space, but are given well-defined interfaces in C++. Choice has not addressed the problems of distributed computing or security. The Clouds system[2] has focused on building a distributed system with large grained persistent objects to address problem of reliability using transactions. Objects provide a uniform paradigm. Extensibility of software components was not a goal. Clouds is programmed in several languages including C, C++, Modula2 and Eiffel.

2 The Spring Object Model

The Spring Object Model includes three aspects of objects. We first describe the "client" view, that is, how objects appear and behave for those that are using them. Next, we describe the "implementation" view, that is,

how an implementation is constructed to abide by the model. Finally, we describe subcontract, a technology for specifying representations and protocols that support the model. Note that it is common (but not required) for a component to be both the implementation of some object(s) as well as a client of some object(s).

2.1 The Client View of the Spring Object Model

By *client*, we mean any component that can possess an object. A client of some object might be the implementation of some other object, but is not required to be. An *object* is an instance of an interface. An *interface* specifies a set of operations that may be performed on the object. An *operation* is a transfer of control that can include the sending or receiving of a set of parameters. *Parameters* may be objects or data values (integers, strings, and simple data structures). An *attribute* is a data value that can be accessed via a pair of implicitly defined operations to store and retrieve the value.

In Spring, interfaces are expressed in an *interface definition language*.[1] The interface definition language is not a programming language, in that it is only declarations. The definitions are mapped into particular programming languages, and the mapping specifies how objects are stored, invoked, and specified as parameters by programs in that language.

The following example shows an IDL definition of an interface,

```
// IDL
Interface NamingContext {
    attribute User owner;
    void lookup(copy Name n,
                produce Object o);
    void bind(copy Name n,
              consume Object o);
};
```

and a C++ mapping of an invocation:

```
// C++ mapping of an invocation
NamingContext context;
File f;
Name filename;

context->bind(filename, f);
```

The type system and semantics of invocation are intentionally simple in order to permit a variety of languages to be used to access or implement Spring objects. Basically, the language needs some way to represent an object, and some subroutine mechanism that allows parameters. Languages with the concept of an object can probably map Spring objects to a constrained usage of the language-defined objects. Languages without such a concept can probably mimic Spring objects with a simple convention.

Objects are passive. Threads are the active entities, which can invoke operations on objects. A (thread in a) client of an object can invoke operations specified by the interface the object supports, or can pass the object as a parameter to an operation on some other object, but cannot otherwise manipulate the object. In order to invoke or pass an object, a client must first *possess* the object. A clients may be given objects when it is created, may be passed objects by being invoked, or may be returned objects as a result of invocations it performs. A client cannot declare an object into existence, nor create an object arbitrarily.[2] In general, clients obtain objects by way of operations on objects, and possession of an object gives the client the right and ability to invoke any operation defined by that object.

The Spring object model is strongly typed. The type of an object is defined by the complete set of interfaces it supports (due to inheritance, a object may support more than one interface as explained below). Each operation of an interface specifies the types of its formal parameters. An object passed as an argument to an operation must support the interface of the formal parameter.

The interface definition language supports inheritance,[3] as shown in the following example:

```
Interface IOStream {
    void get(produce Buffer b);
    void put(consume Buffer b);
};
Interface File: IOStream {
    void seek(copy Position p);
};
```

1. The original name for our interface definition language was *Contract*. The version of this language adopted by the Object Management Group is called *IDL*.

2. Only the implementation of an object can create an object. See next section.

3. Note that this is inheritance of interfaces, and does not imply whether or not inheritance is used in implementations.

```
interface PropertyList {
    void setProp(copy PropName pn,
                 consume Value v);
    void getProp(copy PropName pn,
                 produce Value v);
};
interface FileWithProperties:
    File, PropertyList { }

interface Printer {
    void print(consume IOStream s);
};
```

Instances of the **File** interface support the **get** and **put** operations defined in the **IOStream** interface, as well as the **seek** operation. Multiple inheritance is also possible, as shown by **FileWithProperties**, which supports all of the operations of **File** as well as **PropertyList**. The type of the object is the one interface that includes all the operations that are supported by the object. An object may be passed as a parameter if it supports the interface specified by the formal parameter type. In this example, instances of any of **IOStream**, **File**, or **FileWithProperties** may be passed to the **print** operation.

Parameter Passing Modes

Figure 1 shows a client that contains several objects. By invoking an operation on object O and passing object p as a parameter, O comes to possess object p. The exact details of how the object is transferred is determined by the parameter *mode*.

In ordinary programming languages, it is common for parameter passing mechanisms to have ambiguous usage. For example, passing a variable by reference is used for purely *in* parameters that are too expensive to copy, for *out* parameters, and also for *in-out* parameters when the caller is expecting to use the argument after the callee has changed it in some way; that is, pass-by-reference does not indicate whether a value will be extracted from or inserted into the variable or both. More complicated issues such as whether the callee can retain a reference to the parameter or whether particular operations are considered to modify the object (for example, it may be permissible to increment the reference count, but not to change the value) are usually not specified by the parameter mode.

In programming languages, the pass-by-reference mode is based on shared memory. How could this mode be interpreted for distributed objects? While one may interpret pass-by-reference as passing references to remote object servers, the approach does not work well when you consider passing references to lightweight data-like objects (unless you have shared memory in the distributed environment). One would prefer to pass such data-like objects by value. Choosing between the pass-by-value and pass-by-reference modes for a formal parameter based on such implementation properties (data-like vs. remote) is not possible because such properties are not specified in the parameter type's interface. Indeed, a particular type may have object instances of both kinds.

In a distributed system, it is important to specify which "direction" a parameter is going. There is no reason to transmit a variable to a remote site in order to have a value placed in it. Similarly, if the target is in the local address space, it would be desirable to avoid making copies and get the same benefit as pass-by-reference.

Because we are dealing with objects, in Spring, the five parameter modes that define what happens between the *invoker* and *invokee* have an almost physical intuition to them.

- *Consume*: The argument object moves from the invoker to the invokee at the start of the operation. The invoker no longer has the object.

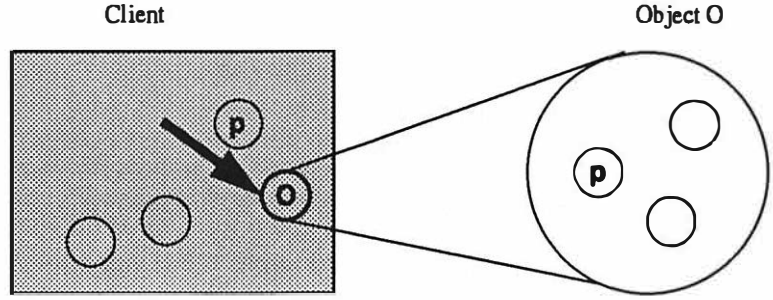


Figure 1. Passing object p as a parameter to object O during invocation

- *Produce*: The argument object moves from the invokee to the invoker at the end of the operation. The invoker has a new object.
- *Borrow*: The argument object moves from the invoker to the invokee at the start of the operation and an object (supporting the same interface, but not necessarily the same as the actual argument object) moves from the invokee to the invoker at the end of the operation. Borrow is equivalent to a consume and a produce of the argument. If the original argument is not returned, the invoker receives a new object in its place.
- *Copy*: A new object is created by calling the standard copy operation on the argument object, and this new object is passed consume. The invoker keeps the original, but no longer has the new object.
- *Share*: A new object is created and passed similar to *copy*, but the assumed semantics of the new object is that it shares behavior with the original.

The consume mode is specified for a parameter for the case when the invoker has generally no interest in using the object after the invocation, and the produce mode is specified when the invokee has generally no interest in using the object after the operation is completed. Consume and produce are two common parameter passing modes and we have found them to be natural and efficient for many interfaces. For example, in the IOStream interface above, the get operation returns the data as a produce parameter and the put operations take the data as a consume parameter.

The consume mode is the most appropriate for the put operation since the client is generally not interested in keeping a copy of the Buffer. In the few occasions when a copy is needed, the client simply makes an explicit copy of the Buffer. Similarly during a get operation, the server (say a file server implementing the IOStream) does not generally need to keep the Buffer. Indeed, we have found that for out parameters, the invokee does not generally want to keep a copy of the parameter and that the produce mode is very appropriate

Borrow is used when the invoker does not need to use the object for the duration of the operation, but wants the object back at the end of the operation. The name object (a data-like object implemented on the client side via a library) provides an equal operation that takes a borrow parameter name to be checked for equality:

```
interface Name {
    boolean equal(
        borrow Name to_be_checked
    );
    ...
};
```

The equal operation checks for equality and returns the parameter unchanged. Consume would not be appropriate since the client would generally want to keep the parameter object and the copy mode would cause an unnecessary copy.

In some uses of the borrow mode, the borrow parameter returned is different from the one passed in, so that the invoker is exchanging one object for another. For example, the interposition service takes an object as a borrow parameter, which is kept by the service, and a interposing object is returned in its place. The interposing object will intercept calls (for tracing, debugging etc.) and then forward the calls to the original object. Although borrow can be emulated using two separate parameters, one consume and the other produce, the explicit declaration of borrow is useful in specifying interfaces.

The copy and share modes are used when the invoker wants to continue possessing the object. In the case of copy, a copy of the object is made on the invoker side and the new object is passed in the same way as consume. The two objects may or may not share state and behavior depending on the semantics of the object. For example, an object of interface File has the meaning "an open file". Passing it copy gives the invokee access to the same file. The objects are different (closing one would not close the other), although their semantics are related (they share the same file data). Although the model does not specify the distinction, for some objects, both copy and share make sense. For example, a buffer might be passed copy to give access to the data and share to allow insertion of data. In case of a file, passing a file copy could mean "shared data but disjoint seek pointers" whereas passing a file share could mean "shared data and shared seek pointer". The additional share mode adds nothing to the model, but supports a common intuition.

There are similar semantics for using the above parameter passing modes for primitive types like integers, reals, strings, etc. and also for constructed types such as structs and arrays. We compare the relative costs of the parameter passing modes in Section 3.4.3.

Object versus the Object Reference

Spring has no separate notion of "object reference". This is important for security and implementation flexibility reasons, as well as to simplify the model. A client

either has a particular object (which was given to it by someone else who had it, etc.), in which case it can do whatever that object can do, or it does not have it, in which case it can do nothing to the object. It is not necessary for the object to protect itself from arbitrary clients who might have forged the object. By eliminating the "object reference" vs. "object" level of indirection, implementations of some objects can be directly in the client. Because objects are abstractions, it is always possible to introduce additional levels of indirection when needed.

Many systems, such as OMG's CORBA[17], make it explicit that the client holds an object reference that points to the object. Nonetheless, because it is designed to support interoperability between object systems and languages, CORBA is general enough about what an object reference is that the Spring object model can be viewed as an extension to it.

To a large degree whether the client is considered to hold the object or a reference to the object is simply a matter of terminology. The Spring object model allows the object state to reside on the server side, the client side, or split between the two. In the case where the entire object state is in a server and the client side simply contains some form of a pointer to the server, the notion of an object reference may make sense. However, in other situations, where the entire or part of the object state resides on the client side, the notion of a reference would have been somewhat misleading and it is more convenient to view the client as possessing the object itself rather than a reference to the object.

We chose our terminology because we wanted the client's view to be that of having abstract objects that can be manipulated and passed around very much like the objects in many object oriented languages.When one writes Spring applications in a programming language like C++ on top of Spring, Spring objects are represented by local C++ objects which fits naturally with the view that the client holds local objects.

Identity

One feature of some object systems that is explicitly avoided is the notion of "identity" for objects. Given that objects are abstract, it is not possible for the object model to determine when two objects ought to be considered the same. An arbitrary rule could be implemented, but it would produce surprising and inconsistent answers. For example, two file objects might be replicas of the same file data. For users of files, we would like to mask the detail of which replica is being used, but for the component that is responsible for updating the replicas coherently, we clearly must distinguish them. Rather than specifying a notion of identity in the object model, we use operations on the objects themselves to answer the question. For example,

```
Interface Document {
    boolean sameas(copy Document d);
    ...
};
Interface ReplicatedDocument: Document {
    boolean sameReplica(copy Document d);
};
```

Constructors and Factories

The model does not provide constructors or the ability to declare an object into existence. Although the model does not permit clients to create objects arbitrarily, conventional usage makes it easy to get access to objects when needed. We call an object whose principal function is to create other objects a *factory*. Factories are widely used for commonly created objects, and language mappings can implement constructors that make calls on the default factories. Access to factories can be controlled like access to any other object, so clients cannot get a factory that creates an object they are not authorized to use.

The most common source of objects is the name service. Operations on a naming context object (similar to a directory) check to see if the client is allowed access to the named object before returning it.

2.2 The Implementation View of the Spring Object Model

The client view of the object model is independent of the choices made in the implementation. Nonetheless, there are important implications about the behavior of implementations that support the object model.

Usually, the same implementation code and associated state are used to implement several similar objects. We call such an implementation an *object manager*. Object managers generally fit into one of two categories: co-resident and server. Co-resident object managers exist in the same domain with their clients. A *domain* has an address space and is the unit of protection and resource management by the operating system. Generally, the operating system does not protect one part of a domain from the other. Server object managers can be protected from their clients by the operating system, and can even be on a different machine. Co-resident object managers can be efficient, but are insecure and typically do not outlive their clients; server object managers are more

expensive, but are safer and can have an independent lifetime.

Although interfaces and objects define the logical boundaries of the software, the enforcement boundaries may be different. When a client possesses an object, it is really held by its domain. Any thread in the domain can in theory invoke operation on objects held by the domain or pass such objects as parameters. An object can be held by only one domain at a time. Domains are also the unit of distribution across machines, that is, all of a domain is on the same machine.

Although the client of a co-resident implementation could in principle refer to the object's implementation directly, it could not do so for a server implementation. In general, the client of an object has a *representation* of the object in its domain, which provides the necessary information for directing invocations to the object's implementation. The representation is typically transmitted from one client to another when an object is passed as a parameter. Client code is unaware of the details of the representation and the actual location of the state and the implementation.

As shown in Figure 2, there a logical distinction between the representation of the object, which usually is close to the client and invariant, and the state of the object, which usually is hidden from the client and manipulated by operations on the object. It is important to realize that although this distinction is often true in practice, it is not a requirement. A category of objects called "server-less" objects have co-resident object managers and the representation is the state of the object. The simplest form of a server-less object is the data object. A *data object* has an interface that consists only of attributes. Specialized implementations merely store the attribute values, and use a co-resident implementation to set or get them.

Because objects are abstractions, there is no complete specification of the state of a particular object. The state of a file, for example, might include disk space management information that is intermixed with that of other files.

A common reason for using a server object manager and distinguishing a representation is to accomplish sharing. For example, two clients may each have a file object that directs invocations to a shared state and implementation in some file server. Other reasons include the need for geographic separation, security, independent lifetimes, etc.

An important implementation technique is to use an object in the domain of the client to implement the mechanism for communication to the object manager. This scheme of creating a "surrogate object" is not novel, but the object model helps ensure that the same mechanism that is used to access the surrogate object can be used for co-resident objects. The simplest form of a co-resident object is the data object.

Whether co-resident or server, the object manager can assume that incoming requests come from clients that are intended to have the object. Directly or indirectly, the object manager creates all the objects it implements. Many object managers accept requests from the name server to (re)create objects in response to lookup requests. These are objects that correspond to persistent state that has existed as objects earlier, and were bound to names. Other object managers create their own factory objects and publish them when they initialize.

Interface versus Implementation Inheritance

A number of object oriented systems and programming languages provide implementation inheritance. *Implementation inheritance* is the composition of an object implementation using parts of another implementation. In some systems or languages, inheritance of an interface implies the inheritance of its implementation. This is not true in Spring: if several interfaces are related by inheritance then Spring does not require that there be any such relationships between the corresponding implementations. We use interface inheritance to structure the system interfaces and to permit substitutability of a derived interface where a base interface is expected. Implementation inheritance is concerned with sharing of code. A Spring implementation must provide all the operations supported by the object, whether in inherited interfaces or not. It may compose that implementation using shared libraries, using implementation inheritance

Figure 2. Client and Server of an Object

with languages such as C++, or use a disjoint implementation as it sees fit. Two closely related implementations may actually share code, whereas two radically different implementations of an interface may have disjoint implementation and would have found any forced sharing burdensome. Thus, while we do not disallow implementation inheritance (and use it in parts of the system), we do not tie it to interface inheritance so as to allow implementations of different parts of the system to change and evolve as needed.

2.3 Subcontract: Programmable Representations and Mechanisms

It may be surprising that we have described the model of objects from both the client and implementation perspective without revealing the representation of objects. This is because in Spring we support a variety of representations, and have defined a means for programming additional ones.

The basic representation in Spring is a door. A *door* is a primitive object used for making protected, remote procedure invocations to another domain, possibly on another machine. Doors are capability-like objects that cannot be forged. A door is implemented, protected, and controlled by the Spring microkernel[8]. A door may be passed as a parameter on an invocation of another door. A domain can create a door that refers to an object implementation within the domain itself. However, a domain cannot create a door that refers to an object implementation in some other domain; such doors must be obtained directly or indirectly from the target domain via other doors. Doors are not persistent—a client cannot store a door in persistent memory for later use. When a client dies, all its doors become invalid. When a server dies, all doors targeted to objects implemented by the server also become invalid.

A door is a plausible representation for an object, and is in fact used that way for many objects in Spring. However, for some objects a door may not be a suitable representation. Although invoking a door is quite fast, for some objects, it is not fast enough (it may want to use a different representation to facilitate some client side caching). Although doors are as robust as most operating system mechanisms, for some services, it is not robust enough. Some objects require a more persistent representation. The tension between performance and simplicity versus security and functionality caused us not to choose a single representation for objects, but rather to allow a variety of them to fit within the same model.

A *subcontract* is a definition of a representation and corresponding invocation protocol [9]. An object implementation selects a particular subcontract to use for its representation. Clients are always unaware of the subcontract used for the objects they possess. Object managers may or may not care which subcontract is used.

In addition to defining the representation and invocation protocol for the object, the subcontract also defines how the object is marshalled and unmarshalled as a parameter on a call on another object. Unmarshalling is interesting (although not to the topic at hand) because it must locate the appropriate subcontract which might not be at the destination [9].

A simple, common subcontract is the single door subcontract whose representation is a single door for each object. A table-entry subcontract could have a representation of a door plus an index into the table. Different table entries could all use the same door, reducing the microkernel's overhead of having many doors. Since the client could lie about the table index, this representation is appropriate only when it is not necessary to separately protect different table entries. The invocation protocol would pass the index as an additional parameter, and the object manager would know about the particular subcontract, and use the index to identify the particular object being invoked.

A persistent-door subcontract would have a representation of a door plus a name that could be used to reacquire the door. If the server crashes, or the door is revoked, the client would go to the name service to get an equivalent door. It is likely that some authentication would be necessary if the object needed to be protected. This procedure could be completely hidden in the subcontract, making it so that neither the client nor the implementation was aware that this particular representation had been chosen.

In many ways, subcontract challenges the Spring object model as much as it helps support it. Exotic subcontracts that support replication, more powerful security, caching, and other properties help to demonstrate we have indeed separated the interfaces from the implementations, at the same time that they help make it practical to use a single model for all of the software. Without subcontract, we would be mired in the traditional conflict between the desire to have pervasive functionality, which argues for putting it in at a low level, and the desire to have a simple efficient system, which argues for leaving it out. With subcontract, we can provide such properties—often transparently—for those objects or in

those environments that warrant them, and omit them where they are not needed.

3 Using the Spring Object Model to build the Spring System

The Spring object model has been used to build the Spring operating system, and has been the basis for our work in Object Products at SunSoft. The system continues to evolve as we reconcile it with emerging standards and transfer the ideas and technology into products.

3.1 The Interface Definition Language, IDL

In the early part of the Spring project, interfaces were specified in Contract. In submitting interface definition technology to OMG[4], we altered the syntax to be closer to C++ and omitted features such as the parameter passing modes[5] that we could not convince OMG to accept. We have also extended IDL to include formal specifications in interfaces that are used for testing (see [20] for more details).

3.2 Using Spring Interfaces

After an interface is specified in IDL, it is translated to mappings for different programming languages: client-side stubs for applications that will use objects of the interface type, and server-side skeletons for implementations of the interface. The actual mapping depends on the programming language. For C++, each IDL interface and its operations are mapped to a C++ class; the methods are client-side stubs, and the C++ object contains the representation of the Spring object which is used to direct calls to the implementation. Primitive and constructed types are also mapped to suitable C++ values.

When a domain starts, it is given some standard objects along with the program argument objects passed by its parent. From these objects it can obtain other objects. For example, one of the objects is a name context through which other objects can be looked up. Some of the objects are factories, for example, a domain uses its domain object to create threads.

A domain can also implement and become a server for an object. Servers make their objects available through the name service or factory objects. For example, a printer server will bind its printer object in a well-known part of the name space for its clients. The kernel provides a factory object called a domain manager that used for creating domains.

Spring also allows lightweight data-like objects where the state and implementation of the object are local so that local procedure calls are used when operations are invoked. A domain can create such an object locally (its implementation is usually available in libraries). When such an object is passed across address space boundaries, the state of the object is passed and the receiving domain must have access to the implementation. For example, names used by the name service are implemented as data-like objects. Another example is the data parameter, called raw_data, of the read and write operations of the IO interface.

3.3 Strong Typing

The Spring object model and the system we have built with it is strongly typed. All arguments to interfaces are statically typed checked. We rely on the static type checking of programming languages: an interface is mapped to a procedure in a high level language such as C++, which performs compile type checking of arguments. Dynamic checking at runtime is performed occasionally. A client always has some perception of the type of every object it holds. The object may of course be some subtype of the perceived type, in which case the client can narrow to this subtype. At this point, the type is checked and the narrow may fail or succeed. For example when a client resolves a name, it expects the object returned by the name service to be of a particular type; the object is returned in some generic form and runtime type checking is performed to convert it to the desired type.

3.4 Experience

Interfaces, inheritance, and the separation of interfaces and implementation have proved very useful in building and extending the Spring system over the last four years. Over that time, we have refined and evolved a style and conventions for using inheritance to make the system open and extensible. A microkernel-based distributed operating system has evolved naturally out of using interfaces to define software boundaries. Unlike many other systems, we take advantage of optimizations that are possible when components are put in the same address space. Building distributed system software has

4. OMG IDL was also influenced by DCE IDL.

5. OMG IDL uses "in", which is copy for data values and share for objects. It calls borrow "inout" and produce "out", and does not yet provide the equivalent of copy for objects or share for data values.

proved to be easier than in conventional systems; however we need further experience with distributed applications. Our experiences with our parameter passing modes have been positive. This section briefly describes our experience in these areas.

3.4.1 Interfaces and Extending the System

The Spring system is made up of a number of components, each with a well-defined interface. Spring is focused on providing good interfaces rather than simply on providing implementations. The interfaces define and capture the various system abstractions. We have spent a fair amount of effort in defining good interfaces to the virtual memory system, the name service, the file system, etc. We allow the coexistence of radically different implementations of a given interface within a single system. For, example, all the following implement the naming context interface: persistent name server, transient name server, the file system, gateways to foreign name servers such as NIS, and the system configuration data base.

Many interfaces are related by inheritance. This has helped us structure the system abstractions so that common abstractions are captured and reused. The classic example is the IOStream interface, which is used by various objects that perform IO (The UNIX operating system exploits a similar abstraction to provide uniformity[21]). Another example is authentication; many of our objects are authenticated, and what it means to be authenticated is captured by the authenticated interface which is inherited by many interfaces. We call such an interface a mixin interface. Other examples of such mixin interfaces are caching and ACLs.

Interfaces have played a key role in extending and evolving the system. Over the last four years several implementations have changed and many interfaces have been extended. The fact that components have well-defined interfaces and are separate from implementations has made it easier to change implementations while insulating clients and localizing the impact of such changes.

Inheritance has proved useful in extending interfaces. For example, we initially defined the file interface and implemented a file system. Later, we decided to add caching. We defined a new interface called cacheable file, which inherited from file, and extended our implementations to support caching. The client's view of file did not change. This is a natural consequence of using inheritance and applications using object oriented programming languages have taken advantage of this in the past; operating system interfaces can also benefit from this and we take advantage of it widely in our system.

We have also developed a scheme for versioning interfaces: different versions of an interface are represented as different types with an inheritance relationship that minimizes the impact on existing clients and allows easy management of versions[10].

Our style of inheritance has evolved to make the system open. Rather than use the classical root inheritance where the system ends up with a few key interfaces inherited by all interfaces, we use a combination of root and leaf inheritance: fundamental properties are inherited early, near the root of the type lattice, whereas auxiliary properties are inherited late, near the leaves of the type lattice, by only those implementations that support the auxiliary properties. This provides flexibility in supporting auxiliary properties, and allows us to add new auxiliary properties as the system evolves without forcing the system to be recompiled [10]. Recompiling the entire system is a real problem in commercial systems where different components are provided by different vendors. Furthermore, shutting down and bringing up a new version of a system is nearly impossible in a distributed environment.

3.4.2 Interfaces versus Address Space and Machine Boundaries

Interfaces are used to define boundaries of software components; these can be mapped to different address space and machine boundaries. The interface definition language has carefully avoided features such as pointers which rely on shared memory which would have prevented components from being distributed across address and machine boundaries. A client of an object is generally unaware of the location of the object's implementation and can perform operations on the object and pass the object around freely.

Writing distributed applications in Spring is easier than in conventional RPC based systems. While the remote procedure call in conventional systems provides transparency in invoking a procedure, distributed programming in such systems is fairly difficult. One of the problems is that the target of a conventional RPC is not a first class entity: a server that is an RPC target has to register itself in some directory service, and clients have to select a target as part of a binding step before performing RCP. In particular, conventional RPC targets cannot generally be passed around as arguments.

In Spring we program using objects which can be passed around as first class entities. The RPC binding step is not necessary[6]. As a result, ordinary client-server interaction is simplified, and more sophisticated distributed computing is significantly easier than in conventional RPC systems. For example, in traditional RPC systems, passing a pointer to a piece of shared data is difficult. In Spring the client can simply implement and create an object that encapsulates the data and pass it as an argument. Similarly in traditional RPC systems, pointers to procedures cannot be passed, unless they made to be full fledged RPC targets that are registered in some directory service. In Spring objects can be passed instead.

Consider the implementation of caching for files in the Spring system. The cacher creates call-back objects on the fly and passes them to the file server for call-backs. Such objects do not have to be registered in any directory service and are simply discarded when no longer needed[7]. Another example is a service that returns a list of entities on some operation. If the list might be arbitrarily large, the service usually returns an iterator object, which can then be used to obtain the list in smaller pieces. Unfortunately, this is inefficient for the common case of small lists. In Spring one can simply return a data-like object that has a small number of entries and, if there are any more entries, the data-like object can contain within itself an iterator that can get additional entries from the server when requested. Such programs are fairly straightforward to write in Spring. The work on network objects at DEC SRC has also noted that objects simplify distributed applications when compared to traditional RPC systems[3].

Our microkernel has been a natural outcome of defining interfaces for the core operating system abstractions, such as address space, processes, virtual memory, file systems, naming, etc. In Spring optimizations are possible when components are placed together in the same address spaces. Components that are configured to reside in the same address space can use local procedure calls and pass arguments directly, avoiding the RPC and marshaling. Furthermore, as explained in more detail below, our parameter passing modes avoid further copying of arguments that would have been necessary for more traditional parameter passing modes. Thus, we can configure our microkernel operating system to reside in a few address spaces and have the same advantage of local procedure calls in monolithic systems.

Subcontracts have played an important role in giving control over the basic object mechanisms and have provided the flexibility needed in supporting features such as caching and reconnection, which are useful in a distributed systems. More detail of subcontract are given in [9].

3.4.3 Parameter Passing Modes

Consume, produce, and borrow modes are efficient since they allow the object to be passed directly without creating new ones; this is especially important when passing object in the same address space where a pointer is passed with no marshalling. The cost of copy depends on the amount of state to be copied; typically for an object with a remote server it means duplicating a door.

In examining the relative costs of different modes we need to consider two things. First, we consider the target object on which the operation is performed; parameters are passed to and from this object. Whether the target is in the local address space or remote matters because in the remote case the parameters needs to be copied to a marshall buffer anyway for all modes. Second, we consider the actual parameter objects. For example a data-like object with a large state has a large representation that needs to be passed—it can be copied or simply passed "directly".

Let us compare the consume and copy modes (the cost of the borrow and produce modes are similar to that of the consume mode). We will see that the consume mode is more efficient than the copy mode, especially when the target object is local.

For a remote target, the parameters are marshalled into a marshal buffer, and the two parameter passing modes are comparable in performance for data-like objects and for primitive data types since in both cases the data has to be copied. If the parameter is a remotely implemented object, the object representation on the client side contains one or more doors (network handles) pointing to the remote site. Thus for the copy mode, doors in the client side object representation need to be copied. This overhead is small since marshalling a door copy can be made to be almost as efficient as marshaling a door consume. Spring also, allows a flexible copy semantics for objects whereby an object's server is notified every time a copy is made. Thus, if a parameter object requires such copy semantics, then the copy mode requires an

6. Although many objects are made available through the name service, others are accessed directly. Indeed, objects are passed to and from the name service for binding and resolving names using the standard parameter passing modes.

7. M. Nelson who implemented the caching file system in both Sprite and Spring found implementing the Spring version significantly easier than the Sprite version which used traditional RPC for the reasons given above.

additional RPC; generally, most objects do not require this

However, when the target object is in the local address space (a data-like object or an object whose server is in the same address space), the consume mode is more efficient than the copy mode, since the parameter can be passed directly and does not need to be copied. The copy cost can be substantial if the parameter object is a data-like object with a large state. For parameter objects that are remote, the copy mode has the overhead of a kernel call to copy the door. Produce and borrow also avoid any copying for operations on locally implemented objects. For example, names are data-like local objects, and the equal operation on names does not have to copy the borrow parameter. Thus, although we can not assume shared memory in the general case, our parameter passing modes give us the same benefits in the local case.

When the target is remote, the copy mode is more efficient than performing a copy operation followed by passing the copied object consume because copying of data takes place anyway during marshalling. We therefore provide two subcontract methods for marshalling: marshal_consume and marshal_copy.

The parameter passing modes have turned out to be natural and easy to use. They were motivated by our need to support distributed programming without relying on shared memory. They give the same advantage as pass-by-reference in the local case.

The basic semantics of parameter passing modes in Emerald[12] is pass-by-reference. An Emerald programmer can specify special modes called *call-by-move* and *call-by-visit* based on his knowledge of the application to optimize invocation; *the semantics remains pass-by-reference* as these modes are simply hints to the system so that it can move the parameter objects if possible.

This approach is different from Spring's where each mode has its own semantics. In Emerald's call-by-move, the caller keeps a reference to the object, where as in Spring the caller does not have the object after it is passed consume[8]. Since for some objects, such as serverless objects, copying the object may have a nontrivial cost, Spring's consume and produce modes offers performance advantages over Emerald's modes.

In Emerald, an object implementor typically specifies call-by-visit if the implementation will be making frequent calls to the arguments during the call. An implementor rarely specifies call-by-move. The caller can also specify the modes based on its knowledge of its own and the server's usage; this is fairly common[11].

In contrast, in Spring, an interface writer specifies the mode based on the direction of the parameters and also on whether the callee of the interface would want to keep a copy of the object.

Emerald's approach was motivated by object mobility. In Spring, mobility is orthogonal to the parameter passing modes. However, mobility is provided automatically for data-like objects that are passed consume, produce or borrow since such an object's implementation decides what is actually transmitted when the object is marshalled.

3.5 Status of the Spring System

Spring currently exists as a fairly complete prototype that is in daily use as a desktop operating system for its own development. The operating system is based around a minimal kernel, which provides basic object-oriented inter-process communication[8] and memory management [13]. Functionality such as naming, paging, file systems, etc. are all provided as user-mode services outside of the basic kernel. The system also provides enough UNIX operating system emulation to support standard utilities such as make, vi, csh, the X window system, etc. All system interfaces are defined in IDL.

The Spring system is distributed to the research community (for details see http://www.sun.com:80/technology-research/spring)

4 Concluding Remarks

The Spring project started with the goal of supporting secure, scalable, easy-to-use, extensible software. Over the last five years we have evolved and used the Spring object model to build a fairly complete prototype operating system. We believe the Spring object model has defined a simple model of security for clients and implementations, a framework for services that span a wide range of performance and capability, a single programming paradigm for data, libraries, system services, and network services, and the means to add, replace, and extend functionality.

8. Spring clients possess objects which can be passed around. There is not a separate notion of a reference to an object.

Interfaces, inheritance, and the separation of interfaces and implementation have proved very useful in building an extensible operating system. The location transparency provided by objects has made writing distributed software easier. Our choice to keep objects abstract with an open representation which can be controlled using the subcontract abstraction has allowed us to use objects at all levels in the system and has provided the flexibility of easily incorporating features such as caching and reconnection that are crucial in distributed environments.

5 References

[1] M. J. Accetta, R.V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian Jr., M. W. Young, "Mach: A new Kernel Foundation for UNIX Development", *Proceedings of the Summer 1986 USENIX Conference*, pp. 93-113, July, 1986.

[2] J. M. Berabeu-Auban, et al, "The Architecture of Ra: A Kernel for Clouds", *22th Hawaii International Conference on System Sciences*, January 1989, pp. 936-945.

[3] A. Birrell, G. Nelson, S. Owicki, W. Wobber," Network Objects", *Symposium on Principals of Operating System*, December, 1993.

[4] R. H. Campbell, N. Islam, P. Madany, "Choices, Frameworks, and Refinement", *USENIX Computing Systems*, 5, 3 (Summer 1992), pp. 217-257.

[5] J. B. Chen, B. N. Bershad, "The Impact of Operating System Structure on Memory System Performance", *Symposium on Principals of Operating System*, December, 1993.

[6] D. R. Cheriton, M. Malcolm, L. Melen, G. Sager, "Thoth, A Portable Real-time Operating System", *Communications of ACM*, 22(2), pp. 105-115, February 1979.

[7] D. R. Cheriton. "The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, pp. 19-42, April 1984.

[8] G. Hamilton, P. Kougiouris, "The Spring Nucleus: A Microkernel for Objects", *USENIX Summer Conference*, pp. 147-160, July 1993.

[9] G. Hamilton, M. P. Powell, J. G. Mitchell, "Subcontract: A Flexible Base for Distributed Programming", *Proc. 14th Symposium on Principals of Operating System*, pp. 69-79, December, 1993. Also available SMLI technical report 93-13.

[10] G. Hamilton, S. R. Radia, "Using Interface Inheritance to Address Problems in System Software Evolution", *Workshop on Interface Definition Languages*, January 1994.

[11] N. Hutchinson, Personal Communication, 1995.

[12] E. Jul, H. Levy, N. Hutchinson, A. Black, "Fine Grained Mobility in the Emerald System", *ACM Transactions on Computer Systems*, 5(1), pp. 109-133 (February 1988).

[13] Y. A. Khalidi, M. N. Nelson, "The Spring Virtual Memory System", Sun Microsystems Laboratories Technical Report SMLI-93-9, March 1993.

[14] Y. A. Khalidi, M. N. Nelson, "Extensible File Systems in Spring", *Proc. 14th Symposium on Principals of Operating System*, pp. 1-14, December, 1993. Also available as SMLI technical report 93-18.

[15] P. Madany, Personal Communication, 1992.

[16] S. J. Mullender, A. S. Tanenbaum. "The Design of a Capability-Based Distributed Operating System", *The Computer Journal*, 29(4), pp. 289-299, 1986.

[17] Object Management Group, "Common Object Request Broker Architecture and Specification", OMG Document 91.12.1, December 1991.

[18] S. R. Radia, M. N. Nelson, M. L. Powell, "The Spring Name Service", Sun Microsystems Laboratories Technical Report SMLI-93-16, October 1993.

[19] S. R. Radia, P. Madany, M. L. Powell, "Persistence in the Spring System," *Proc. 3rd International Workshop on Object Orientation in Operating Systems (I-WOOOS III)*, pp. 12-23, December 1993.

[20] S. Sankar, R. Hayes, "ADL: An Interface Definition Language for Specifying and Testing Software", *Workshop on Interface Definition Languages*, pp. 13-21, January 1994.

[21] K. Thompson, D. M. Ritchie, "The UNIX Timesharing System", *Communications of ACM*, 17(7), pp. 365-375, July 1974.

Integration of Concurrency Control in a Language with Subtyping and Subclassing

Carlos Baquero* Rui Oliveira* Francisco Moura*

Departamento de Informática / INESC
Universidade do Minho
Braga - Portugal

Abstract

This paper describes the integration of concurrency control in BALLOON, an object-oriented language that separates the concepts of type and class as well of subtyping and subclassing. Types are interface specifications enriched with concurrency control annotations. Classes are used to implement the operational functionality of types as well as concurrency control mechanisms. Types, classes and concurrency control annotations are independently reusable and derivable. The language takes advantage of this separation to solve the typical problems of the inheritance anomaly.

1 Introduction

BALLOON is a novel object-oriented language intended as a test-bed for experimenting with concurrency in a object-oriented, potentially distributed setting. It is a compiled, statically-typed language that allows explicit creation and management of concurrency while avoiding the traditional conflicts between inheritance and concurrency.

The management of large-scale software projects calls for separate compilation and joint development through code reuse and incremental development. These features are strongly related to the notions of encapsulation, inheritance and component interface description which are the foundations of object-oriented technology. Therefore, BALLOON not only advocates the separation of subtyping and subclassing but also distinguishes concurrency control classes from operational classes. As a result, types, classes, and concurrency control are independently reusable and derivable, thereby eliminating the inheritance anomaly.

BALLOON is expected to contribute to the design of extensible libraries of concurrent components. In fact, BALLOON allows the derivation of concurrent components without requiring the knowledge of any internal details about its ancestors, as these may well be hidden in a compiled library. It suffices to know the type interface and develop a properly annotated class.

In the next section we briefly overview existing approaches to concurrency in object orientation focusing on the inheritance anomaly. Subsequent sections describe the model of types and classes in BALLOON with emphasis in the language constructs relevant for the following sections. Next, the extensions for concurrency control are introduced showing how they interact with the type and class structures of the language and how they cope with the inheritance anomaly. This is then compared with related work and finally the paper's contributions are summarized.

2 Concurrency in Object Orientation

The introduction of concurrency in the object-oriented model has been intensively investigated in recent years, where numerous alternatives and pers-

*Email: {cbm,rco,fsm}@di.uminho.pt. WWW: http://www.di.uminho.pt/~{cbm,rco,fsm}. The first author was partially supported by JNICT/Praxis XXI BD/3123/94.

pectives where evaluated; these approaches were classified and organized in several taxonomies [27, 14, 21, 6, 23]. A common conclusion stemming from this research is that integration of concurrency and object orientation is not a trivial task.

In the presence of multiple threads, and thus multiple active methods, some mechanisms are needed for regulating concurrent accesses to object data. However, the introduction of concurrency control mechanisms in the object model, can interfere with the inheritance mechanism. The definition of derived components often requires modifications to existing concurrency control code in the supertype, specially when this code is embedded within the operational code. As a result, inheritance is severely impaired unless appropriate abstractions are used for the incremental definition of concurrency control code. This interference is known as the inheritance anomaly [21], and it has been the subject of intensive research on several frameworks [22, 19], particularly on Actor based languages as ABCLONAP1000 [21], ACT++ [15, 14], ROSETTE [31].

2.1 Brief Introduction to the Inheritance Anomaly

In the study of the inheritance anomaly, some examples are widely used to depict specific problems in inter-object [21, 14] and intra-object[1] [22, 30, 18, 6, 4, 28] concurrency control.

2.1.1 Inter-Object Concurrency Control

Explicit Reception (Acceptance) of Messages
Languages such as ADA [12] and CSP [13] adopt bodies that regulate the acceptance of messages on active server objects. On these bodies, possibly guarded accept statements are interleaved within the operational body code. The addition of new methods on derived components incurs on possibly very complex body redefinitions, thus preventing the inheritance of bodies. Some affected languages are POOL-I [2], μC++ [8], EIFFEL// [10].

Accepting Methods According to the Abstract State

Addition of new methods often changes existing relations between the abstract object state and the invocable methods. Languages based on *accept sets* [15, 31] define named entities that describe which methods are invocable at a given moment.

Consider a stack with `push` and `pop` operations. We can identify the following accept sets that describe different abstract states:

State	Methods
Empty	push
Partial	push, pop
Full	pop

However, introducing a method `pop2` that removes two elements from the stack creates a new relevant state, thus partitioning the previous representation as shown below.

State	Methods
Empty	push
Singular	push, pop
Partial	push, pop, **pop2**
Full	pop, **pop2**

Languages that rely on the explicit switching of the current abstract state such as ACT++[2] [15] and ROSETTE [31], are unable to program this example without awkward redefinitions.

The introduction of more flexible mechanisms for the use and extension of *accept sets* as those introduced in the new ACT++ version [14] and in ABCLONAP1000 [21] solves this problem. However, there remains the subtle problem of exposing the implementation details of the concurrency control code. The programmer of a derived component must know the structure of the inherited sets, so as to be able to change them, and include the newly defined methods where appropriate.

Accepting Methods According to History Dependencies
A potential alternative to the *accept sets* is to rely on the definition of method predicates [3, 19, 28], and to control method invocation with boolean guards. This allows rather elegant solutions to the previous problems.

[1] These schemes also apply to inter-object coordination.

[2] The version presented at ECOOP'89.

For example, if `load` expresses the current number of elements on the stack, and `MAX` is its maximum size, we can associate the guards $(load < MAX)$, $(load \geq 1)$, $(load \geq 2)$ to `push`, `pop`, `pop2`, respectively. These guards are mutually independent and can be added along with new methods on derived components.

Unless guards are assisted by additional synchronization actions, the definition of methods that depend on the invocation history of the components is made impossible as the appropriate history information must be recorded. The definition of a method `stat` that should only be accepted after 100 invocations of the other stack methods would require the use and update of a counter. Guards should also be extensible on derived classes, although not necessarily as in [19] where they can only be made more restrictive.

2.1.2 Intra-Object Concurrency Control

Intra-object concurrency can have a significant impact on performance. For example, in a simple experiment carried out on a 2-cpu system, disabling the internal concurrency of a producer-consumer through a bounded buffer reduced efficiency to 57% [5].

Approaches to the inheritance of intra-object concurrency control were evaluated with *accept sets* in [22] and *synchronization counters*[3] in DRAGOON [28], SINA [6] and DOOJI [30]. The use on DEMETER [18] of *exclusive regions* to which methods are associated is, to some extent, similar to the definition of named *accept sets*.

2.2 Concurrency Annotations Control Model

The CA (Concurrency Annotations[4]) model [4] defines a way of introducing separate concurrency control mechanisms for the coordination of multiple threads of execution and protect concurrent accesses to the internal object's state.

[3] Such as the number of threads that are executing a given method.

[4] This term is also used in CEiffel[17] but with a different meaning.

With CA, method invocation is protected by a guard and a lock. Only when the guard evaluates to true and the lock is opened can the corresponding method be allowed to execute, otherwise the calling thread is blocked in a wait queue. Concurrency control actions can be executed before and after method execution. These actions are standard code of the host language and may define and change private state elements of the synchronization code. They also have the special capability of acting on the method locks, and are able to use the interface of the controlled object. Actions and method guards can be independently expanded or redefined on derived components.

Concurrency control in BALLOON uses a modified version of the CA model. These features will be presented after an overall description of BALLOON.

3 The BALLOON Programming Language

BALLOON is a concurrent object-oriented programming language aimed at providing expressive abstractions for the reuse and composition of software components. In some aspects BALLOON resembles POOL-I [2] and SATHER [29] providing the complete separation of types and classes. Types specify the interface of objects sufficient for their correct composition and interaction in a message-based system [23]. Types are implemented by classes, which are the main structuring block of the language. This binding is a many-to-many relationship in the sense that a type can have multiple implementations and a class can also implement several types. The pair of a type and its implementations is usually referred to as a component.

Due to this explicit separation of types and classes, subtyping and subclassing relations are also distinguished concepts as advocated in [11, 24]. Subtyping serves data abstraction and conceptual structuring [16] while subclassing is left unconstrained allowing a flexible code reuse mechanism through inheritance [32, 25].

Genericity in BALLOON is achieved through generic components. Generic components differ from ordinary components by defining particular *nuclear types* on which the component's specification and

implementation depend. The genericity mechanism provides the ability to derive new components by the explicit substitution of the nuclear types of a generic. Using explicit substitutions instead of parameterization, generic types and classes can be classified at the very same level as their ordinary counterparts. This is an important result for the orthogonality of the two code reuse mechanisms [26].

3.1 Type Hierarchy

Types form a well-defined hierarchy ordered by the subtype relation. As usual, because types are interface specifications, subtyping is defined in terms of interface compatibility [32]. A type is said to be a subtype of another if for every method of the later the former provides a method with the same name preserving covariance in the result and contravariance in its arguments [9]. Subtype polymorphism is supported by allowing an object to be substituted, in any context, by objects whose type is a subtype of the former's type.

The type hierarchy of BALLOON is therefore constituted by ordinary and generic types. The main order[5] in the hierarchy is given by subtypes. At the top of the hierarchy there is the type *Any* whose every type is subtype of, and there is also a virtual type, named *Nil*, which is a subtype of any other type in the hierarchy.

3.2 Class Hierarchy

Classes related by inheritance form an hierarchy. There is not any conformance relation between classes related by inheritance. The relation stands solely as a class constructor mechanism. Inheritance is the pure inclusion of code with subsequent re-binding of self references.

Because the inheritance mechanism is unconstrained, allowing multiple inheritance, renaming of inherited methods, and absence of any conceptual ordering, it is likely to yield a multi-topped hierarchy.

[5] Aside from subtyping, other type relations are defined in BALLOON but they are not relevant to this paper.

3.3 Relevant Constructions

Examples of the basic constructions of BALLOON relevant to this paper are now presented[6]. The examples remain simple serving as a base for the extensions that will be presented in the next section.

3.3.1 Types

A type in BALLOON is the declaration of the interface the objects of that type must support. An example is depicted next

```
Type IntQueue
  {
    Integer PutLeft(Integer);
    Integer GetRight();
    Integer Size();
  }
```

A possible subtype of IntQueue is an IntDEQueue (a double ended queue of integers). Its definition may be

```
Type IntDEQueue SubtypeOf IntQueue
  {
    Integer PutRight(Integer);
    Integer GetLeft();
  }
```

which means that IntDEQueue has all the methods of IntQueue unchanged plus the PutRight and GetLeft methods. As the received methods of IntQueue remain unchanged they are, by default, contravariant in their arguments and covariant in their results.

3.3.2 Generic Types

Container types (for example sets, bags, queues, etc) are more likely to be defined as generic types from where specific ones might be derived. The next example defines a generic DEQueue.

```
Type DEQueue
  {
    BasedOn T <: Any;

    T PutLeft(T);
    T PutRight(T);
    T GetLeft();
    T GetRight();
    Integer Size();
  }
```

DEQueue can hold any kind of objects because it is based on Any. Being based on Any it can be used

[6] A report on the definition of the language is currently being written [1].

to derive any kind of DEQueue. The above IntDEQueue could be obtained by

```
Type IntDEQueue
  DEQueue[Integer/T]
```

Deriving a new type from a generic one can be done by the substitution of a nuclear type by any subtype of the nuclear type.

It should be noted that a generic is itself a complete type. It can be used in any context an ordinary type is because their nuclear types are always bound to existing types. In the DEQueue example, T is bound in the type declaration to Any by the T <: Any expression. Defining a type as ordinary or as generic only depends on the conceptual entity it models.

The derivation of a new type from a generic does not force any type relationship between these types. As with any other types they may be related by subtyping or simply unrelated.

3.3.3 Classes

Classes provide the implementations of the interfaces declared in the types. The binding is made in the class definition. The class declares the type or types it implements. The following is the implementation of the IntDEQueue type. Notice that, in the example, we do not bother to validate any methods in exception conditions such as the full and empty queues. This will be done in the next section using concurrency control extensions.

```
Class CIntDEQueue Implements IntDEQueue
 {
    Data IntArray contents;
    Data Integer(CInteger) left, right, size;

    CIntDEQueue(Integer qsize)
     {
        contents = CIntArray(qsize);
        size := qsize;
        left := 0;
        right := size - 1;
     }
    Integer PutLeft(Integer elem)
     {
        contents.Put(left,elem);
        left := (left + 1) % size;
        return elem;
     }
    Integer PutRight(Integer elem)
     {
        contents.Put(right,elem);
        right := (right - 1 + size) % size;
        return elem;
     }
    Integer GetLeft()
     {
        left := (left - 1 + size) % size;
        return contents.Get(left);
     }
    Integer GetRight()
     {
        right := (right + 1) % size;
        return contents.Get(right);
     }
    Integer Size()
     {
        return size;
     }
 }
```

As IntDEQueue is a subtype of IntQueue its implementation CIntDEQueue is also an implementation of IntQueue. This holds for any type, that is, any implementation of a type is also an implementation of that type supertypes. Any implementation of a type is also an implementation of any of its supertypes.

To show the independence of the inheritance mechanism from type relationships consider the next example. Suppose a type IntStack (a stack of integers) defined as

```
Type IntStack
 {
    Integer Push(Integer);
    Integer Pop();
    Integer Top();
    Integer Size();
 }
```

which has no relation with IntDEQueue (nor with IntQueue). However, a possible implementation of IntStack can be achieved by inheritance of the CIntDEQueue class as depicted next

```
Class CIntStack Implements IntStack
 {
    Inherits CIntDEQueue Renames
        PutLeft as Push, GetLeft as Pop;

    CIntStack(Integer qsize)
     {
        CIntDEQueue(qsize);
     }
    Integer Top()
     {
        var Integer(CInteger) i;
        i := (left - 1 + size) % size;
        return contents.Get(i);
     }
 }
```

CIntStack provides an implementation to IntStack. It inherits from CIntDEQueue, renames PutLeft and GetLeft to match the IntStack type and implements (apart from the constructor) the Top method.

Finally, an implementation of a generic type may be sketched. Implementing the DEQueue type is like writing the CIntDEQueue class but now based on the type Any such as:

```
Class CDEQueue Implements DEQueue
 {
    BasedOn T <: Any;
    Data Array contents;
    Data Integer(CInteger) left, right, size;

    ...
```

that is replacing the relevant Integer type occurrences by T which is the nuclear type of the class.

For each type derived from the generic DEQueue an implementation can be derived from this generic class. As an example consider the following implementation for the IntDEQueue.

```
Class CIntDEQueue
  CDEQueue[Integer/T]
```

An interesting point about substitution instead of parameterization is that it is orthogonal to inheritance. This allows the two mechanisms to be composed, that is, both CDEQueue and CIntDEQueue could be now inherited and extended.

4 Concurrency Control In BALLOON

This section shows how the BALLOON type system is used to document and apply concurrency control policies. The IntQueue type will be enriched with a new section, CC, where some activation clauses and additional methods are defined. These methods are only accessible in activation clauses that are defined in the CC section of the type or in its subtypes. These activation clauses follow the format: **Delay** *method-name* **Until** *predicate*. The activation of the method is delayed until the predicate evaluates to true.

```
Type ConcIntQueue SubtypeOf IntQueue
 {
    CC:
    Integer MayPut();
    Integer MayGet();
    Bool AllowLeft();
    Bool AllowRight();
    Delay PutLeft Until ( AllowLeft() && MayPut() > 0 );
    Delay GetRight Until ( AllowRight() && MayGet() > 0);
 }
```

The two clauses specify in which circumstances should the operational methods PutLeft and GetRight be allowed to execute. As the Size method returns a constant value that is fixed after instance creation, there is no need for an activation clause. These clauses introduce a predicate that uses the operational and concurrency control methods defined in this type or in its supertypes.

Appearing in the type's definition, the CC constraints are meant to characterize the type's allowed concurrency. These constraints act, to some extent, as a specification for all implementations of the type. Although the concrete behaviour of the introduced methods is unspecified, the **Delay Until** clauses restrict the set of implementations of the type. An example of this will be given in section 4.1 after the introduction of the ConcIntDEQueue type.

The newly introduced concurrency control methods have their implementation in appropriate classes. These classes do not directly implement a type as before, but are said to control operational ones. Mixed through inheritance, an operational class and its controller class yield a class that fully implements the concurrent type. The encoding of the activation clauses takes place in any class that implements a concurrent type. This separation of operational and concurrency control classes provides the ability of their independent manipulation and reuse.

The example that follows presents an implementation of the CC methods in type IntQueue. In addition to the class syntax used above, the class CCIntQueue is able to define a wrapper that adds pre- and post-actions to the operational methods which are represented by an inner statement. The syntax is: **Wrapper** *method-name* {*pre-actions*} **Inner** {*pos-actions*}. It is worth to note that *inner* represents only the operational methods for objects of this type. In a subclass the wrapper is inherited and the *inner* represents the heir's own wrapper. This notion of the

inner construct is similar to that found in BETA [20] inheritance mechanism.

These actions are in the scope of the class CCIntQueue definition, and so may only refer to instance variables defined in that class (or in inherited) and to the operational interface of the controlled class.

All concurrency control actions are executed with mutual exclusion in each object. So, only the operational methods are allowed to run in parallel. This also ensures that, if a method invocation successfully passes the activation clause, then the corresponding pre-actions are finished before allowing the evaluation of another invocation.

```
Class CCIntQueue Controls CIntQueue
 {
   Data Integer(CInteger) mayput, mayget;
   Data Bool(CBool) allowleft, allowright;

   CCIntQueue()
     {
       allowleft := true;
       allowright := true;
       mayput := Size();
       mayget :=0;
     }
   Integer MayPut() { return mayput; }
   Integer MayGet() { return mayget; }
   Bool AllowLeft() { return allowleft; }
   Bool AllowRight() { return allowright; }
   Wrapper PutLeft
     { allowleft := false; mayput := mayput - 1;}
     Inner
     { allowleft := true; mayget := mayget + 1;}
   Wrapper GetRight
     { allowright := false; mayget := mayget - 1;}
     Inner
     { allowright := true; mayput := mayput + 1;}
 }
```

With this scheme CIntQueue[7] is regulated for intra and inter-object concurrency. Operations that would mutually interfere are made exclusive, whilst those that are independent are allowed to run in parallel. This behaviour was expressed by the use of the flags *allowleft* and *allowright*. A new class that implements the concurrent type ConcIntQueue is built by mixing the operational class CIntQueue and its controller CCIntQueue as shown below

```
Class CConcIntQueue Implements ConcIntQueue
 {
   Mix CIntQueue, CCIntQueue;
 }
```

The semantics of **Mix** is that of inheritance plus the wrapping of operational methods.

[7] In fact the CIntQueue definition has been omited. It is subsumed by CIntDEQueue.

The coordination of inter-object concurrency was expressed in the type's activation clauses and relied on the current "putable" and "getable" number of elements of the queue.

The next section shows how these mechanisms are integrated with subtyping and subclassing.

4.1 Subtyping and Subclassing with Concurrency Control

The concurrency control information for the type IntDEQueue, subtype of IntQueue, may be defined as follows

```
Type ConcIntDEQueue SubtypeOf IntDEQueue, ConcIntQueue
 {
   CC:
   Delay PutRight Until ( AllowRight() && MayPut() > 0);
   Delay GetLeft Until ( AllowLeft() && MayGet() > 0 );
 }
```

The class CCIntDEQueue is a possible implementation for the concurrency control of CIntDEQueue.

```
Class CCIntDEQueue Controls CIntDEQueue
 {
   Inherits CCIntQueue;

   Wrapper GetLeft
     { allowleft := false; mayget := mayget - 1;}
     Inner
     { allowleft := true; mayput := mayput + 1;}
   Wrapper PutRight
     { allowright := false; mayput := mayput - 1;}
     Inner
     { allowright := true; mayget := mayget + 1;}
 }
```

The use of class inheritance, in this case reusing CCIntQueue, allowed the definition of concurrency control for just two methods, the ones introduced on the subtype IntDEQueue. Possible intra-object concurrency interferences are dealt by excluding operations on the same side of the queue. This is aided by the inter-object coordination that avoids the join of the sides, this would happen if, for example, two opposite get operations where allowed when there is only one element in the queue. The activation clauses and the modification of the mayput and mayget variables on the wrapper actions ensures that this situation does not occur.

The way the activation clauses were written in the ConcIntDEQueue type allows the concurrent execution of left and right methods. Recalling what have been said earlier about the specification role of the

activation clauses, the reader should notice that implementations of ConcIntDEQueue are constrained by them. Only operational classes that do permit the concurrent execution of left and right methods may be used to implement this type. As a counter example, an algorithm based on a global counter of elements that would be incremented by both put methods and decremented by both get methods would not correctly implement this ConcIntDEQueue type.

It became apparent, and results quite naturally, that in most cases the pattern of class inheritance for concurrency control mimics the subtype relationships. However there are some exceptions, as in the type ConcIntStack.

```
Type ConcIntStack SubtypeOf IntStack
 {
    CC:
    Bool Allow();
    Delay Push Until ( Allow() && MayPush() > 0 );
    Delay Pop Until ( Allow() && MayPop() > 0 );
    Delay Top Until ( Allow() && MayPop() > 0 );
 }
```

Concurrency control is trivially implemented by reusing CCIntDEQueue code.

```
Class CCStack Controls CIntStack
 {
    Inherits CCIntDEQueue Renames
      AllowLeft as Allow, MayPut as MayPush,
      MayGet as MayPop;
 }
```

Finally, to what concerns generic types concurrency control, it could be easily shown that since the concurrency control does not interfere with the operational interface of types, the control of derived types can be exactly that of the generic type it was derived of. The same happens for concurrency control of generic implementations. As an example, the reader may notice that the concurrency control of the above IntDEQueue example would be the same of the DEQueue generic type.

4.2 Inheritance Anomaly Examples

Although the implementation of ConcIntDEQueue and ConcIntStack already shows the avoidance of the inheritance anomaly, some additional examples are required to show how the BALLOON approach deals with the anomaly. The previous examples also show that the addition of methods (such as Size) that do not interfere with existent ones is trivial since the default activation clause is: **Delay** *method-name* **Until** (true).

The introduction of a method GetLeft2 (equivalent to Pop2[8]) will show how to cope with changes of the abstract state of the object. Consider the following subtype of ConcIntDEQueue

```
Type ConcIntDEQueue2 SubtypeOf ConcIntDEQueue
 {
    IntegerPair GetLeft2();

    CC:
    Delay GetLeft2 Until ( AllowLeft() && MayGet() > 1 );
 }
```

and the controller class of the CIntDEQueue2 (which for space saving reasons is omitted) is

```
Class CCIntDEQueue2 Controls CIntDEQueue2
 {
    Inherits CCIntDEQueue;

    Wrapper GetLeft2
      { allowleft := false; mayget := mayget - 2;}
      Inner
      { allowleft := true; mayput := mayput + 2;}
 }
```

which delivers a concise description of the synchronization scheme for the new type and reuse all the previous code.

Until now, there was no need to redefine activation clauses or extend actions. The definition of a method GGetRight or GGetLeft that only performs a *get* operation if used immediately after a *put* operation would require the recording of additional information to express history dependencies. The same happens with a Stat method that may only work after 100 invocations.

The following implementation of Stat in a subtype of ConcIntDEQueue shows how this is achieved in BALLOON. From this example its straightforward to infer how to implement GGet operations.

```
Type ConcIntDEQueueS SubtypeOf ConcIntDEQueue
 {
    Integer Stat();

    CC:
    Integer Calls();
    Delay Stat Until ( Calls() > 100 );
 }
```

[8] If we had opted to extend the type Stack the description would be cluttered by irrelevant renaming issues.

```
Class CCIntDEQueueS Controls CIntDEQueueS
 {
   Inherits CCIntDEQueue;
   Data Integer(CInteger) calls;

   CCIntDEQueueS()
     {
       calls := 0;
     }

   Integer Calls() { return calls; }
   Wrapper PutLeft {} Inner { calls := calls + 1; }
   Wrapper GetRight {} Inner { calls := calls + 1; }
   Wrapper PutRight {} Inner { calls := calls + 1; }
   Wrapper GetLeft {} Inner { calls := calls + 1; }
 }
```

Consider, in this example, the wrapper construct of method PutLeft. Here **Inner** refers to the PutLeft method only. This whole wrapper constitutes now the **Inner** of the corresponding wrapper in the inherited controller class. That is, it corresponds to the **Inner** of the PutLeft wrapper defined in CCIntDEQueue.

The expansion in this case is as follows:

```
From CCIntQueue:
[   { allowleft := false; mayput := mayput + 1;}
    From CCIntDEQueue:
    [   {}
        From CCIntDEQueueS :
        [   {}
            Code of PutLeft from CIntDEQueue
            { calls := calls+ 1; }
        ]()
    { allowleft := true; mayget := mayget +1;}
```

These rules provide the ability for concatenation of pre- and pos-actions through the hierarchy of control classes.

A difficult case of inheritance anomaly is brought by the introduction of reusable generic locking mechanisms, generally provided in mixin classes [7]. Mixins are specially designed classes so that when mixed with other classes (usually through multiple inheritance) provide some added functionality.

The type ConcIntDEQueueL introduces three additional methods named AllLock, WriteLock and UnLock. These methods are intended to, respectively, lock all the methods, lock the methods that change the object state and unlock all of the previously locked methods. This type also shows how to redefine (in this case extend) previously defined activation clauses.

```
Type ConcIntDEQueueL SubtypeOf ConcIntDEQueue
 {
   AllLock();
   WriteLock();
   UnLock();

   CC:
   Bool AllowAll();
   Bool AllowRead();
   Delay PutLeft Until ( SuperClause && AllowAll() );
   Delay GetRight Until ( SuperClause && AllowAll() );
   Delay PutRight Until ( SuperClause && AllowAll() );
   Delay GetLeft Until ( SuperClause && AllowAll() );
   Delay Size Until ( SuperClause && AllowRead() );
 }
```

Consider the next two classes. They are a operational class and a control class that act as simple containers of generic code.

```
Class CLock
 {
   AllLock() {}
   WriteLock() {}
   UnLock() {}
 }

Class CCLock Controls CLock
 {
   Data Bool(CBool) allowall, allowread;

   CCLock()
     {
       allowall := true;
       allowread := true;
     }

   Bool AllowAll() { return allowall; }
   Bool AllowRead() { return allowread; }
   Wrapper AllLock { allowread := allowall := false; }
       Inner {}
   Wrapper WriteLock { allowall := false; }
       Inner {}
   Wrapper UnLock { allowread := allowall := true; }
       Inner {}
 }
```

These classes can now be inherited in order to implement the ConcIntDEQueueL type.

```
Class CConcIntDEQueueL Implements ConcIntDEQueueL
 {
   Mix CCLock, CLock, CIntDEQueue;
 }
```

With this scheme the introduction of locks in a given type only requires the extension of the existent activation clauses. The newly created type ConcIntDEQueueL clearly expresses in its signature the new invocation constraints, without being cluttered with implementation details. If necessary, some activation clauses can be defined for methods AllLock, WriteLock and UnLock so that, for example, UnLock is only allowed after a AllLock or WriteLock

invocation.

5 Related Work

The separation of types from classes has already been proposed in object-oriented languages [2, 29]. This is mainly due to the observation that the subtype relation does not always follow inheritance [11]. America introduced these notions in a language with concurrency concerns [2] from where this paper in many aspects has its beginnings. Lately, several researchers [23, 21] point this separation as a promising issue in treating the inheritance anomaly in concurrent object-oriented languages.

In what concerns concurrency control the use of *accept sets* has been a common factor in most of the recent proposals for solving the inheritance anomaly. The amount of auxiliary mechanisms for the definition, extension and sometimes dynamic computation of *accept sets* varied considerably along these proposals. In particular, ABCLONAP1000 [21] introduced multiple mechanisms for the computation of *accept sets* showing how they were applicable to the inheritance anomaly. Recently the DOOJI model [30] studied the extension of some ABCL mechanisms for the support of intra-object concurrency. Coordination of intra-object concurrency relied on *synchronization counters* also used on DRAGOON [28] and SINA [6].

We believe that the default computation of *synchronization counters* can be avoided since, when needed, they are easily programmable in pre- and pos-actions and then used on BALLOON activation clauses. This reduces to "only when necessary" an otherwise fixed overhead.

BALLOON also avoids the use of *accept sets* since they require in some cases, such as when adding a Get2 operation, complex manipulations of the inherited sets (see [21, 30]). These manipulations and renamings are often difficult to track along the inheritance chain. The use of extensible activation clauses together with extensible wrapper actions provides, in most cases, a simpler and more self contained solution.

The separation of concurrency and operational code, mandatory in BALLOON, has been (in different degrees) previously defended on several frameworks [18, 28, 22, 4].

6 Conclusions

The main contribution of this paper is the introduction of a concurrency control model that takes advantage of the language distinction between types and classes to solve elegantly the typical problems of the inheritance anomaly. In particular, components that were designed without any concurrency concerns can be easily extended to fit in concurrent environments. This is achieved by annotating the component's type and designing appropriate controller classes for its coordination. This coordination respects intra- and inter-concurrency control.

The ability to define concurrency control for generic types introduces a new, orthogonal, reuse mechanism. Due to the modularity inherent to the model, concurrency control does not interfere with encapsulation and inheritance.

A similar concurrency control model (section 2.2) has already been tested as an extension of the C++ language, producing code for a shared memory multi-processor architecture with kernel-supported threads [5, 4]. This experience, although in a language with no separation of hierarquies, raised important implementation issues that have influenced the design of the BALLOON model.

A BALLOON compiler comprising the full set of language features described in this paper is under development.

Some issues are still open and under current research. One is the use of the inner construct for the composition of pre- and post-actions. Since the inner actions are nested, there is no easy way to cancel the inherited post-actions. However, such an example is rather unlikely in that it would also probably entail the redefinition of activations clauses. In this case, one would advocate a new control class.

7 Acknowledgments

We thank Paulo Sérgio Almeida for continous feedback on early drafts of this paper. We also thank Laurent Thomas, Cristina Lopes, Birger Andersen, Rachid Guerraoui, Lodewijk Bergmans, Eric Jul and the anonymous referees for their useful comments.

And of course, to the other members of the BALLOON development team, António Coutinho and Victor Fonte for their efforts in bringing BALLOON to life.

References

[1] P. Almeida, C. Baquero, A.Coutinho, V. Fonte, F. Moura, and Rui Oliveira. The Balloon programming language: Specification and Rationale. Technical report, DI - Universidade Minho, 1995. In Preparation.

[2] Pierre America. A parallel object-oriented language with inheritance and subtyping. In *ECOOP/OOPSLA'90*. Springer-Verlag, 1990.

[3] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, and Vandôme. Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1):31–67, 1991.

[4] C. Baquero and F. Moura. Concurrency annotations in C++. *ACM SIGPLAN Notices*, 29(7):61–67, July 1994.

[5] Carlos Baquero. Inheritance of synchronization code on object-oriented concurrent programming. Master's thesis, DI - Universidade do Minho, 1994. In Portuguese.

[6] Lodewijk Bergmans. *Composing concurrent objects.* PhD thesis, University of Twente, June 1994. ISBN 90-9007359-0.

[7] Gilad Bracha and William Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90*. Sringer-Verlag, 1990.

[8] P. Buhr, G. Ditchfield, R. Stroobosscher, and B. Younger. μ C++: Concurrency in the object-oriented language C++. *Software-Practice and Experience*, 22(2):137–172, February 1992.

[9] Luca Cardelli. Semantics of multiple inheritance. In *Semantics of Data Types*, LNCS 173. Springer-Verlag, 1984.

[10] Denis Caromel. Toward a method of object-oriented concurrent programming. *Comunications of the ACM*, 36(9):90-102, September 1993.

[11] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *POPL '90*, January 1990.

[12] G. Goos and J. Hartmanis. *The Programming Language Ada Reference Manual.* LNCS 155. Springer-Verlag, 1983.

[13] C.A.R. Hoare. Communicating sequencial processes. *Communications of the ACM*, 21(8):666–477, 1978.

[14] Dennis G. Kafura and R. Greg Lavender. Concurrent object-oriented languages and the inheritance anomaly. In *ISIPCALA'93*, 1993.

[15] Dennis G. Kafura and Keung Hae Lee. Inheritance in actor based concurrent object-oriented languages. In *ECOOP'89*. Springer-Verlag, 1989.

[16] Barbara Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5), May 1988.

[17] Klaus-Peter Lohr. Concurrency annotations for reusable software. *Comunications of the ACM*, 36(9):81–89, September 1993.

[18] Cristina Lopes and Karl Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In *ECOOP'94*. Springer-Verlag, 1994.

[19] Svend Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *ECOOP '92*, pages 185–196. Springer-Verlag, 1992.

[20] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Addison Wesley, 1993.

[21] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. Research Directions in Concurrent Object Oriented Programming, MIT Press, 1993.

[22] Christian Neusius. Synchronizing actions. In *ECOOP '91*. Springer Verlag, 1991.

[23] Oscar Nierstrasz. Composing active objects – the next 700 concurrent object-oriented languages. In *Research Directions in Concurrent Object Oriented Programming*. MIT Press, 1993.

[24] Rui Carlos Oliveira. Subtypes and subclasses in the object-oriented paradigm. Master's thesis, DI - Universidade do Minho, 1994. In Portuguese.

[25] Jens Palsberg and Michael Schwartzbach. Three discussions on object-oriented typing. *ACM SIGPLAN OOPS Messenger*, 3(2):31–38, 1992. Summary of ECOOP'91 Workshop on "Types, Inheritance and Assignments.

[26] Jens Palsberg and Michael I. Schwartzbach. Type substitution for object-oriented programming. In *OOPSLA/ECOOP'90*. Springer-Verlag, 1990.

[27] Michael Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming.* PhD thesis, Dept. of Computer Science, University of Geneva, 1992.

[28] S. Crespi Reghizzi, G. Galli de Paratesi, and S. Genolini. Definition of reusable concurrent software components. In *ECOOP'91*. Springer Verlag, 1991.

[29] C. Szyperski, S.Omohundro, and S. Murer. Engineering a programming language: The type and class system of Sather. LNCS 782. Springer Verlag, November 1993.

[30] Laurent Thomas. Inheritance anomaly in true concurrent object oriented languages: A proposal. In *IEEE TENCON'94*, pages 541–545, August 1994.

[31] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled-sets. In *OOPSLA '89*. ACM, 1989.

[32] Peter Wegner and Stanley Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP'88*. Springer-Verlag, 1988.

Generic Containers for a Distributed Object Store

Carsten Weich
Institut für Informatik, Universität Klagenfurt
Universitätsstr. 65, A-9020 Klagenfurt, Austria
e-mail: carsten@ifi.uni-klu.ac.at
www: http://www.ifi.uni-klu.ac.at/cgi-bin/staff_home?carsten

May 18, 1995

Abstract

In this paper we report of an experiment about how generic container classes can be used to build up a distributed main memory object store. This approach is inspired by the C++ standard template library (STL) but uses set-like associative structures instead of the array-like sequential structures mainly used by the STL. The basic structure is a container with only fundamental functionality. The only way to access the data of the elements is to apply a function to all the elements. We show how such containers can be extended to indexed sets and distributed object stores.

We use Modula-3 to implement the generic classes. Experiments show that such containers form powerful building blocks especially suitable for implementing distributed containers.

1 Overview

Object stores and object oriented databases usually use the file system as main storage area. Operations on such databases are designed to limit the number of accesses to disc blocks. Sometimes they use large main memory areas as buffer to accelerate operations. In [1] we have presented a different approach: Why not use main memory as main storage area and use the file system only to back up data. This should make it much easier to implement database operations. Since main memory is limited on a single compute node, we make use of distribution and parallelism in order to scale well with a growing database. If you add a compute node to your database you not only get more room for storage but also additional computational power.

We want to use sets as our main container type. Sets are not ordered by definition—this makes it much easier to parallelize operations automatically. The programmer must not make assumptions about the order of performing operations on elements of a set [2]. Consequently the operation can be parallelized by applying it on distributed subsets. Initial observations presented in [3] have shown that with this scheme you can scale up the size of a set without making operations slower by adding compute nodes to the system. The additional communication time needed can be made up by the additional computational power from the new nodes.

The main part of this paper presents generic containers and a generic distributed object type. They will be used to implement sets. We express them in Modula-3 [4]. With generic programming, container structures can be developed independently from the structure of the elements. The standard template library [5, 6] has shown that it is possible to go even further: You can develop algorithms independently of the container structures they work on. This is possible by designing one basic generic type with which container structures can be built. In case of the standard template library this basic type is an abstract notion of a pointer. As we will explain in the second section of the paper, we choose a different approach. Our basic type is less low level, we use whole containers as basic types.

Our containers provide a very simple interface with basic operations, which can be implemented efficiently. You can build up a variety of data structures by subtyping containers and by combining different kinds of containers. Due to the simplicity of the interface, they can be used as object store, as index into objects stores or even as building blocks for more complex, distributed structures. As examples we show how indexed containers and distributed containers can be built using our basic container classes.

We show how *distributable objects* can be made using containers. Distributable objects have an object identifier which is independent of the address space they currently reside. They can be accessed from remote compute nodes, and they can move from one node to another. Again we use generic code to develop such objects.

Finally we show how this approach meets the requirements of our distributed object store architecture. We report about our experiments and give an outlook on the future investigations planned.

Goals

We want to achieve a distributable object store. In order to make parallelization of operations on the stored data simpler, we use sets as storage structures. A large set can be distributed by installing subsets of it on several object store nodes. Many basic operations can be parallelized easily by just applying them on these subsets in a first step. Then the results of the sub-operations (which are hopefully much smaller in size) have to be collected in a second step. Selecting objects that meet certain criteria, calculating sums, maximums or averages and many more fall in that category. So distributable sets are needed as our basic structure.

On the other hand, many clients need to access data according to a certain ordering. This is not possible in sets. So sorted indexes into the sets are needed. They can be used to iterate through the set. Indexes are also needed if a client wants to access the data by means of a key value. Obviously indexes must be distributed as well if they become large.

Let us summarize the required structures: We need set structures for small, large and huge sets. For sets with frequent update accesses as well as for sets which are only readed. And we need sorted index structures for all these. There is no data structure to meet all the needs. We will have to provide a variety of data structures the client can choose from.

We can do this by developing generic building structures, which can be combined to support certain requirements of a particular client. Let us take a look at a well tested library of generic data structures first:

2 The Standard Template Library

The Standard Template Library (STL) [5] was designed to provide generic building blocks for general purpose programs. It also provides set containers. Indexed containers are not directly supported but we could certainly develop generic code to build them.

In order to construct a unified view to every supported data structure, STL defines a set of operations with which all data structures are accessed. Together these operations are called *iterator*. This leads to generic code which not only takes the element type as a parameter. It also parameterizes the whole data structure. Thus, algorithms working on a certain structure will work for a broad range of different structures as well. The operation to switch from one element to the next in the structure must be built in the iterator, not into the algorithm. The iterator must also provide a way to return the element itself (called *dereferencing*). The element must provide a way to compare its value with another element.

STL algorithms see containers as pairs of iterators (*first* and one beyond *last*). They access the structure by iterating through it by means of the predefined operations. By this, STL algorithms see every data structure as an array. Some algorithms are only allowed to iterate through the array by advancing from on element to the next, some algorithms may jump back or forth to the *n*-th element of the array. It is the job of the implementor of iterators to provide this view. Linked lists, trees or files can be mapped to iterators. A sorting algorithm for example works for arrays as well as for a file of records—the iterators are pretty different, the names and parameters of the operations accessing the data is always the same.

We prefer to consider sets as the most primitive view on aggregations of data. Instead of iterating through the elements, functions are applied to every single element—in an undefined order. By this, direct access to every element is encouraged, sequential processing is discouraged. We hope that this view will lead to parallelizable datastore operations. It is not easy to adopt this view to the STL. Since iterating through the structure sequentially is STL's most basic operation, it is not easy to develop a distributed structure.

But we can make use of the ideas behind STL to develop a special purpose generic library for distributable object stores. We also have to transfer the STL mechanisms to Modula-3 [4], because this is our implementation language. The generic Modula-3 containers presented here have a different view than the STL. We do not distinguish between the abstraction of iterators and containers. By not specifying how containers are processed, we make parallelization much simpler.

3 Generic Containers

Containers are data structures which store elements of a certain kind. You can insert and remove elements. You can apply a function to all elements stored. You can also test whether an element is in the container or not (fig. 1). Some implementations may rely on hints about the maximum or average number of elements the container can hold. You can pass this number when initializing the container. A container can tell the number of elements it currently stores.

Set oriented containers are associative in a sense that they only provide access to already known elements. You can not identify elements by position nor can you scan from one to the next. You can search elements by stating

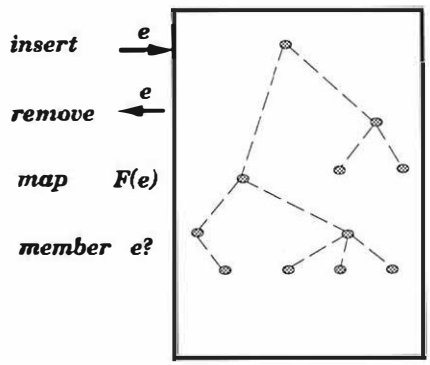


Figure 1: The basic structure

a search function which is applied to *all* elements. If you already have an element, you can test, whether it is in a certain container or not.

We define table containers which store tuples (key, element pairs) as well. If you insert an element into a table container, you also have to provide the elements key value. You later access the elements via this key value.

3.1 The Interfaces

We need two basic interfaces, the associative containers called `Cntr` and the table containers called `TableCntr`. The first one looks like the following[1]

```
GENERIC INTERFACE Cntr(Super, Elm);
  TYPE T <: Public;
    Public = Super.T OBJECT METHODS
        init (sizehint: CARDINAL := 0): T;
        member  (e: Elm.T): BOOLEAN;
        insert  (e: Elm.T);
        remove  (e: Elm.T);
        map     (p: Closure);
        size    (): CARDINAL;
      END;
    Closure = OBJECT METHODS
      apply(e: Elm.T)
    END;
END Cntr.
```

Table containers are defined by the interface:

```
GENERIC INTERFACE TableCntr(Super, Elm, Key);
  TYPE T <: Public;
    Public = Super.T OBJECT METHODS
        init   (sizehint: CARDINAL := 0): T;
        member (k: Key.T): BOOLEAN;
        insert (k: Key.T; e: Elm.T);
        remove (k: Key.T);
        get    (k: Key.T): Elm.T;
        map    (c: Closure);
        size   (): CARDINAL;
      END;
END TableCntr.
```

[1] We do not list the exception handling and other details here.

Both `Cntr` and `TableCntr` are generic interfaces. In Modula-3 this means, that they have to be instantiated with explicit interfaces. The parameters are the super type of the container, the type of the elements and the keys. The module defining the element or key type must export a type `T` and at least two of the following procedures:

```
PROCEDURE Hash (e: T): Word.T;
PROCEDURE Equal (e1, e2: T): BOOLEAN;
PROCEDURE Compare (e1, e2: T): [-1..1];
```

These procedures define how different instances of elements or keys can be compared with each other. `Hash` returns a hash code which can be used to identify elements and to store them in hash tables. The other two are used to compare the value of instances. Which ones are needed in a certain case depends on the implementation of the container. For modules describing types, it is a common convention in the Modula-3 library [7] to provide these procedures.

Containers can be used in very different ways. Implementers might want them to be subtype of `Netobj.T` [8]—to install the container on a remote server—or of `MUTEX`—to be able to lock the container to synchronize multiple accesses to it. So not only the element type is a parameter but also the super type.

Applying functions

The way to access the elements of a container is to apply a function to all of them. The function is passed to the container with the `map` method. This method takes a `Closure` as parameter which contains the function (as `apply`-method). If the function needs parameters or if it produces a return result, the `Closure` must be subtyped. The customized closure contains parameters and return fields as attributes. For instance a function calculating the sum of the `salary`-fields of all `Person.T` elements of a container could be applied using the following closure:

```
SumCl = Closure OBJECT
          sum:= 0;
        OVERRIDES
          apply:= SumOfSal;
        END;
```

This closure can be passed to the `map` method of the container, the `SumOfSal` procedure can access the `sum` field, which will contain the final result after `map` has terminated. The same mechanism is used in connection with threads in Modula-3 [4].

As another example let us define a closure for selecting all `Person.T` objects that have a `salary`-field greater than a certain value:

```
SelectCl = Closure OBJECT
             value : CARDINAL;
             result: PersonCntr.T;
           OVERRIDES
             apply:= SelectPersons;
           END;
```

The value is passed to the `SelectPersons` procedure with the `value` field. The procedure will test each element individually and insert those meeting the condition into the `result` container.

3.2 Different kinds of containers

The exact specification of the container operations are left to the implementor of the generic interfaces. E.g. we might have containers allowing the insertion of duplicates (like in bags) and others, which will ignore such insertions (like in sets). For simultaneous access to containers by different clients: Some implementations might require the client to lock the whole container before any update action, others might lock individual elements.

Very often it is necessary to store the data in a sorted fashion. For this purpose we define subtypes of the basic container classes. The sorted associative container class looks like the following, the same technique can be used for table containers:

```
GENERIC INTERFACE OrdCntr (Cntr, Elm);
  TYPE
    Direction = {Left, Right};
    T <: Public;
    Public = Cntr.T OBJECT METHODS
        init (sizehint: CARDINAL := 0): T;
        iterate (first: Elm.T;
             dir := Direction.Right): Iterator;
      END;
    Iterator = OBJECT METHODS
        next (VAR e: Elm.T): BOOLEAN;
      END;
END OrdCntr.
```

We do not want to change the semantic of the `map` method: The order in which it is applied to the elements is still not defined. To access the elements sequentially, we provide an additional `iterator` method. It returns a cursor object which can be used to scan through the elements of the container in the order defined by the `Elm.Compare` function. Note that code written for basic containers can still be used for ordered containers, because the latter are subtypes of the first.

We will use subtyping for a number of other purposes as we will see in the following.

3.3 Subtypes of the Container Classes

The operations performed by containers are rather fundamental. There is only the `map` method to access the elements, e.g. for tasks like searching or working on subsets. Algorithms which are of interest for more than one client are implemented as generic modules which define subtypes of some container type. Obviously this means that they do not have to be reimplemented for whatever container they should work on. As first example let us look on set operations. A generic module containing the common set operations could have an interface like the following:

```
GENERIC INTERFACE Set(Container, Elm);
  TYPE  T <: Public;
    Public = Container.T OBJECT METHODS
        init   (): T;
        select (c: SelectClosure): T;
        union  (s: T): T;
        intersect (s: T): T;
        equal     (s: T): BOOLEAN;
      END;
    SelectClosure = OBJECT METHODS
        test (e: Elm.T): BOOLEAN;
      END;
END Set.
```

The interface is called `Set`, but if the super type is a container that allows multiple insertions it will act like a bag. The key point is that the (generic) implementation of the `Set` module uses only operations of the container interface.

The method `select` returns a new instance of the same set type container, which contains all elements that have met a certain criteria. The criteria is tested by a `test` method passed to `select` as parameter. The methods `union` and `intersect` can be used to generate new sets containing the union or the intersection between the set itself and another set passed as parameter to the methods. Finally `equal` can be used to test whether the set contains the same elements as another one passed as parameter. All these can be implemented using the `map` method defined in the super type. If you want to print out all elements of the set that meet a certain condition, you write:

```
set.select(cond).map(print)
```

Where `set` is an instance of type `Set.T`, `cond` provides a `test` method checking the condition and `print` provides the method that prints individual elements.

3.4 Rearranging containers

If you need alternative access to some container data, or if you want to rearrange the data it contains, all you have to do is to copy it. E.g. sorting a container means to copy it to an ordered container. A trivial generic module will do this job:

```
GENERIC INTERFACE CntnrCopy (Cntnr1, Cntnr2);
  PROCEDURE Copy(c1: Cntnr1.T): Cntnr2.T
END CntnrCopy.
```

Its generic implementation only needs the map method of the source container and the insert method of the destination.

4 Putting containers together

Now we are able to show the efficiency of our containers. Due to the simplicity of their interfaces, it is rather easy to use them as building blocks of more complicated structures (which might them self be containers). We will present two examples: Containers with indexes attached to them and distributed containers. Both use containers again to implement their features. So it is possible to control their behavior, efficiency and complexity by passing suitable container instances as parts when instantiating the final structures. For instance you can pass a distributed container as index structure to the generic indexed container—which will lead to a container with a distributed index without one single additional line of code.

4.1 Indexed Containers

One of the most important structures in large data stores are indexes. They provide alternative access to large amounts of data to accelerate certain repeatedly needed retrievals. Suppose we have a huge container containing all student data records of an university. If a certain algorithm needs access to all students attending a course we need an index pointing to only that data. Another need might be to access the student data by their name and alternatively by their "matriculation number".

Using generic containers this is a very simple task. We first define a subtype of a generic container class containing additional methods to attach and detach indexes to it. Note that it is left to the instantiation whether the super type is a ordered container or not (see 3.2).

```
GENERIC INTERFACE IndexedCntr
                  (Cntr, Index, Elm, Key);
  TYPE
    GetKey = PROCEDURE (e: Elm.T): Key.T;
    Test   = PROCEDURE (e: Elm.T): BOOLEAN;
    T <: Public;
    Public = Cntr.T OBJECT METHODS
        init (sizehint: CARDINAL:= 0): T;
        addIndex   (index: Index.T;
                    getKey: GetKey;
                    test: Test:= NIL);
        removeIndex(index: Index.T);
      END;
END IndexedCntr.
```

The method addIndex attaches a new index structure to the container. Index.T is a table container mapping Key.T values to Elm.T instances (which are the elements of the indexed container). You have to provide a getKey procedure which calculates the key value of a particular element. As an option you can also pass a test procedure which decides whether an element should be contained in the index or not.

So if we want a structure containing all students attending a course, we do the following: We provide a container for that structure, pass it to the indexed student set using the addIndex method together with a test procedure (which selects the corresponding students). This will generate the index and the indexed container will keep it up to date until we detach it with the removeIndex method.

You can add more than one index to an indexed container. Still all have to have the same key type. If this is not convenient, you can subtype an indexed container again with the same generic interface but a different key type parameter. The subtype will have two addIndex methods, one for each key type.

At this point we list a fragment of the generic code of the indexed container. It is the implementation of the insert method.

```
PROCEDURE Insert (self: T; e: Elm.T) =
  BEGIN
    Cntr.T.insert(self, e);
    VAR ind:= self.indices;
    BEGIN
      WHILE ind # NIL DO
        IF ind.test = NIL OR ind.test(e) THEN
          ind.index.insert(ind.getKey(e), e)
        END; (*IF*)
        ind:= ind.next
      END (*WHILE*)
    END
  END Insert;
```

The statement Cntr.T.insert(self, e) is a super call to insert the element in the container itself, the line ind.index.insert(...) updates the index. This is all we have to do. The complete generic implementation of indexed containers counts little more than 100 lines of code.

4.2 B-Trees

B-trees and B*-trees were developed to minimize access to disc blocks when searching data identified by a key. While balanced binary trees perform very well in memory, it makes sense to store more than one key in a tree node if accessing the node is expensive [9]. The same is true if the data is distributed among several compute nodes (see fig. 2). Since the amount of data stored in a compute node will be much larger than in a disc block, we need a special kind of B-tree: The root tree node is much smaller than the tree nodes in the machines. The nodes containing the data can be huge. Looking up elements on a single node must also be efficient.

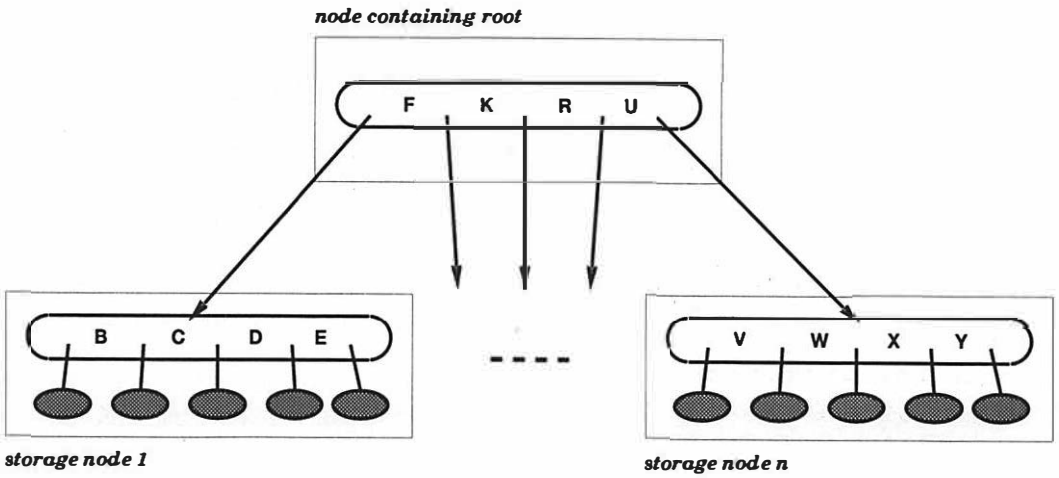


Figure 2: A distributed B*-tree object store

Again we try to develop such a structure using containers as building blocks. We need three kinds of containers:

- *Node containers*
  They store the elements of the distributed container. They must be able to split themselves: This generates a new node if necessary, when the amount of data increases. They also have to be able to merge themselves with another node. If the amount of data decreases, several nodes can be combined in this way.

  The split and merge methods can be implemented using the `map` method of the generic containers. So any container can be transformed into a node container.

- *Root node*
  This is a table mapping elements to node containers. It should be a sorted table in which the smallest element (according to some element ordering) stored in a node is mapped to a pointer to the node containing it.

  The root node can be implemented with our basic table containers.

- *Superstructure*
  We call the structure containing the root table container-container. It is a subtype of some container class with additional methods (see below).

  The implementation of a container-container is responsible for the distribution strategy. The client can choose between different implementations to get containers which distribute the elements over equally sized substructures, or to get one which replicates the data to accelerate certain client operations, and so on.

Let us take a look at the container-container interface:

```
GENERIC INTERFACE CntrCntr (Cntr, Elm);
  TYPE
    T <: Public;
    Public = Cntr.T OBJECT
      METHODS
        init(maxPerNode, minPerNode,
             sizehint: CARDINAL:= 0): T;
        map (fct: Closure;
             coll: CollectClosure:= NIL);
      END;
    CollectClosure = OBJECT METHODS
        collect (cls: ARRAY OF Closure);
      END;
END CntrCntr.
```

We have to redefine the `init` method in order to pass additional initialization information to the distributed container: How many nodes will be available? At what size does the client want to split a node into two? What is the minimum size of a node, i.e. when does the client want a node to be merged with others?

The `map` method needs a second closure: The first one defines the function which has to be applied to all elements (in all sub nodes). Since we want to do this in parallel, we get several provisional results. These can be computed to a final result with the `collect` method of the second closure. It can be left nil, if there is no such result.

As we have seen it is possible to build a variety of storage structures with the container interface (and with little changes to it). A formal description of a certain instance of a container must be supplied by the implementor. It

must define the operations exactly and describe their complexity. A system of generic implementations of various kinds of containers is being developed at our department.

5 Generic Distributable Objects

The generic containers support access to arbitrary elements. They do not address the problems of accessing fields and methods of remote stored objects. Especially they do not solve the problems of simultaneous access to the data of a particular element. In this section we demonstrate how access to remote objects can be organized using containers. The task of looking up the location of a remote object and retrieve its data is very similar to the task of retrieving an element from a container. If we could express the access to remote objects using container mechanisms, we could make use of distributed containers to implement remote data access.

5.1 Dereferencing with Methods

In Modula-3 objects are identified by their reference, which points to their memory address. This is not suitable for objects which can reside on several compute nodes. Especially path expressions are difficult to implement if the location of the data is not known. Think of a person record storing the car the person owns. The car record might have a reference to the manufacturer of the car, which is a record containing the name of the company. Determining the car builder of the person's car would require an expression like the following:

```
person.car.manufacture.name
```

The person record, the car record and the manufacturer record might reside all in different address spaces. Further on, an assignment like

```
person.car := bmw_525;
```

which is called when the person buys a new car, again might require write access to several storage nodes. This is not possible if the objects above are ordinary Modula-3 objects.

Before we can access the object's data, we have to insure that we have an up-to-date version of it. This could be done by calling a method before reading data. The above path expression should look like:

```
person.r().car.r().manufacture.r().name
```

The `r` method checks if the data is locally available. If not it has to retrieve it. Finally it returns the actual value of the data. Note that changing this value should not change the actual value of the object.

The expression for setting the persons car attribute should look like:

```
person.w().car := bmw_525.r();
```

The effect is that the `w` method requests write access to the data object. It returns a reference of the data's physical location. This reference points to a writable location of the data which now can be changed.

The Modula-3 library contains the so called *network objects* [8], which provide the possibility to access remote objects. Network objects can be used to implement containers on remote nodes. But they were mainly designed to provide access to remote *services*. It is not possible to read or write object fields directly, you can only call the object's methods. And then, network objects were not made to support hundreds of thousands of objects, which might change their location frequently.

So we propose a different scheme. We need a construct which can be used instead of object references. The following interface describes a distributable object pointer:

```
GENERIC INTERFACE DistObj (Container, Elm);
  IMPORT Word;
  TYPE T <: Public;
     Public = MUTEX OBJECT METHODS
       init(c: Container.T; e: Elm.T): T;
       copyref(): T;
       r(): Elm.T;
       w(): Elm.T;
     END;
  PROCEDURE Equal(o1, o2: T): BOOLEAN;
  PROCEDURE Compare(o1, o2: T): [-1..1];
  PROCEDURE Hash(o): Word.T;
END DistObj.
```

Obviously this interface is suitable as element type for our containers—it contains a `T` and the necessary comparing procedures. This type can be used instead of a main memory address pointer. Dereferencing a distributable object can be done with two methods: `r` dereferences for reading, `w` dereferences for writing. Obviously, read and read/write access can be granted to clients by showing all or only part of the distributable object's interface. The `r` method returns the actual *value* of the object. The `w` method returns a pointer to the a writable location of the object. If the contents of this location is changed, the actual value of the object changes.

When initializing a distributable object, we pass the objects data and a container to the `init` method. The container serves as an abstract representation of the physical location of the object.

The `copyref` method returns a pointer to the object, not the object's value. This representation is of the same type and points to the same instance than the original. It is not necessary to look up the current value of the object. The method acts like assignment of pointer values. It is not necessary in a single address space. But the representation returned by `copyref` can be sent to another node.

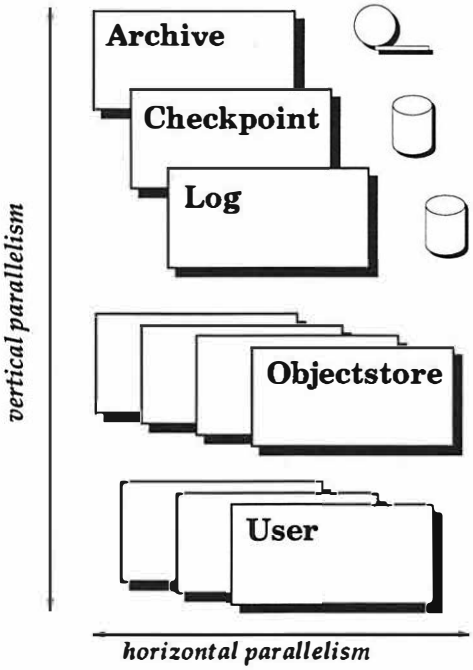


Figure 3: The components of PPOST

5.2 Implementing Distributable Objects

In a distributed object store with simultaneous access of several clients, we have to deal with the following problems.

1. We have to get an actual version of the data when the `r`- or `w`-method is called,
2. We have to deal with locking and consistency problems if simultaneous read and write accesses occur. If there are several copies of an objects values around, we have to ensure consitency after every write access.

The physical location of the objects data is represented by a container. You pass the container which can be used to retrieve the data when initializing an instance of a distributable object. The code implementing the `DistObj` interface delegates the problem of physically distributing the data to this container. It concentrates on implementing the locking scheme and on keeping consistency.

By using containers as abstraction for physical locations, we are able to separate the two main problems of remote object access. Keeping consistency is not a problem which a container can solve. But it can solve the problem of retrieving the elements. Different locking strategies can thus be combined with different storing solutions.

6 Containers for PPOST

The architecture of PPOST was presented in [1, 3]. In this section we would like to show, that the presented generic container scheme can be used to implement such an architecture. PPOST's main components are (figure 3): *object store* (consisting of a number of object *storage machines*), *log machine*, *checkpoint machine*, *archive machine* and *users* (consisting of a number of user machines). All the data of the stored objects (i. e. their attributes and methods) lie in the memory of the storage machines. PPOST is transaction-oriented. Every transaction that reads or changes the data is executed on those machines. Transactions are initiated by the user machines and processed by the object store. Changes of the data in the object store are reported to the log machine which saves the information onto a log file in non volatile memory. This can be done by for instance by writing to a sequential file with maximum disc speed.

The checkpoint machine reads the log produced by the log machine and saves all committed changes to the disc-based database. It produces a structured image of the database. If this requires more time, only the log file on the log machine will grow. The user transaction can go on as soon as the information about the changes is transmitted to the log machine.

The archive machine saves the disc-database to a secondary storage, like a magnetic tape. This is considered as a normal activity of the data-store and again is done in background without interrupting the user-transactions.

We call this pipeline-like way to decouple user-transactions from issues of persistence *vertical parallelism*. Operations on the stored data can often also be done in parallel, we call this *horizontal parallelism*.

With our container templates we are able to offer several structures to organize the object store. The users will get containers as super structures. With our distributable objects we have explicit control over all write accesses to the data: All users have to use the `w` method described in the last section. This method has all the necessary information to produce the log records.

7 Results and Future Work

Generic programming turned out to be an efficient technique to implement many variants of object stores. Once you have decided to view your data in an uniform way, the solutions you find for a particular problem can always be reused for many more problems. Since the parts which are developed all have similar interfaces, they can be tested easily and partly automatically. Thus more complicated structures are made with robust building blocks. Modula-3 proofed to be suitable for this programming

philosophy.

The technique is especially powerful when building distributed object stores. It makes it easy to switch between different internal structures say for node subsets or indexes. Since the result of the combination of sub containers again forms a container, the technique can be applied recursively. Once we have a distributed container, we also have distributed sets, indexes or even nodes to super-super structures.

Because of this results, we are currently working on library of different generic containers (it is available via links from the authors WWW home page). They seem to provide a very flexible framework for experimenting with different distribution strategies.

8 Related Material

This paper, the Modula-3 code of the generic containers and the PPOST papers are available via world wide web. They can be accessed via links from the authors home page (the www-address is listed at the beginning of the paper).

References

[1] L. Böszörmenyi, J. Eder, and C. Weich. Ppost, a parallel database in main memory. In *Proceedings of the Fifth International Conference on Database and Expert Systems Applications*, 1994.

[2] L. Böszörmenyi and K. H. Eder. Adding parallel and persistent sets to modula-3. In *Proceedings of the Joint Modular Languages Conference*, September 1994.

[3] L. Böszörmenyi, K. H. Eder, and C. Weich. Ppost, a persistent parallel object store. In *Proceedings of the Second International Conference Massively Parallel Processing Applications and Development*, 1994.

[4] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[5] A. A. Stepanov and M. Lee. The standard template library. Doc no: X3j16/94-0095, wg21/n0482, ISO Programming Language C++ Project, 1994.

[6] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software--Practice and Experience*, 24(7):623–642, July 1994.

[7] Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. Some useful modula-3 interfaces. Research report 113, Digital Systems Research Center, Palo Alto, 1994.

[8] A. Birell, G. Nelson, S. Owicki, and E. Wobber. Network objects. Research report 115, Digital Systems Research Center, Palo Alto, 1994.

[9] R. Sedgewick. *Algorithms in Modula-3*. Addison-Wesley, 1993.

Media-Independent Interfaces in a Media-Dependent World

Ken Arnold
Ken.Arnold@east.sun.com
Sun Microsystems Labs
2 Elizabeth Dr.
Chelmsford, MA 01824

Kee Hinckley
nazgul@utopia.com
Utopia, Inc.
25 Forest Circle
Winchester, MA 01890

Eric Shienbrood
ers@wildfire.com
Wildfire Communications
20 Maguire Rd.
Lexington, MA 02173

Abstract

Wildfire is a communications assistant that uses speech recognition to work over phone lines. At least that's what it is today. But in the future it wants to run on desktops, PDAs (like the Newton Message Pad), and who knows what all. To provide a level of media independence, we designed a subsystem to isolate the communications knowledge of the assistant from the mechanisms of prompt/response. This layer is called the MMUI. It provides abstractions of input and output that let the assistant ask questions and get responses without knowledge of the specifics of the communication channels involved. The specifics of speech recognition, as well as the degree of abstraction desired, make this an interesting case of presentation/semantic split using object polymorphism. This presentation will cover the design of the MMUI, its fundamental weaknesses, and furious handwaving over future directions to mend them.

1. Introduction

The Wildfire communications assistant is designed to use computer analysis and assistance to enhance communication, both with other Wildfire users and with the outside world. To do this, the interface is critical: it must be natural and easy to use, engaging without wasting your time, and so on. There is nothing in any reasonable requirement list that says it must *only* work over voice interaction, and in fact, future expansion pretty much demands that it be more flexible than that, for example to operate with tty lines for the deaf, text-based two-way pagers, and eventually pen-based PDAs and GUI-based desktops.

However, much of the value added of the system has nothing in particular to do with the presentation of the interface. The primary value is provided using knowledge of how and when to get in touch with a person, who has called and when you need to call them back, who is currently on hold and who is important, and how to weave these facts into more effective assistance to the user.

The concept of a presentation/semantic split in application design is well established [1,2,3,4,5], so it was obvious that a presentation layer needed to be provided for developing the Wildfire assistant. This layer, however, has several additional requirements:

- It needs to be able to handle completely linear interactions. A voice interaction is like a conversation, in which the assistant asks questions, waits for a response, and then either gives feedback or asks another question.
- Voice interactions have a different quality to the interaction than a GUI does. As an example, because of the nature of speech recognition, a choice from a menu is not a single selected thing, but a list of probability-ordered *possible* responses.
- In any single interaction, user input can come from a variety of source. For example, you can either speak your responses, or you can use touchtone shortcuts.
- Speech recognitions systems can require training, resulting in a significant collection of user-specific data associated with the general menu.

So beyond the normal presentation/semantic considerations, there was a need to interact with the user independent from the specific ***media*** through which the interaction was taking place. Presentation could be through recorded voice, text-to-speech, plain or internationalized text, or a GUI presentation mixing images and text. Input could be recorded voice, recognized voice, text, images (such as faxes) and so on.

The independence of the need for interaction from the media through which the interaction takes place, and the

great value represented by the underlying communications enhancement independent of the interface, made it attractive to create an abstraction capable of isolating the assistant code from the particular media(s) through which the user was interacting.

We also added the following requirements:

- It must be easy to add new types of media. It is impossible to predict who will make the next advancement in price or functionality in speech recognition, or to predict the winner in the two-way pager marketplace. Adding new kinds of media should require relatively minimal work that is completely hidden from the assistant.
- It should be possible to select particular media based on other attributes. While providing a layer of abstraction for the variability of media, adding a generic variability mechanism to select for, say, the desired prompt verbosity seemed like a small addition for a large gain.

2. The MMUI

The abstraction we designed is called the MMUI (*M*ulti-*M*edia *U*ser *I*nterface). It lets the assistant interact with the user through a very detached abstraction using units of meaning.

(For those of you who want to follow along with pictures, Figure 1 shows the class hierarchy for most of the classes described in this paper.)

The basic meaning abstraction is the ***Meme***. When the assistant wants to say "Hello", it doesn't care if the presentation is in text, voice, or video, or even what language is used. What matters is that the user is presented with representation of the concept of "Hello" that is meaningful to them.

Each representation of the concept "Hello" does, in the end, have to be presentable in some way via a specific representation understood by a particular device, such as an audio board or an ASCII stream. Specific presentable data are represented by ***Media*** objects. The media class is an abstract base class from which specific media representation classes are derived.

The "Hello" meme would thus contain several media objects of various types, such as a TextMedia object containing the string "Hello" and/or an AudioMedia object that describes a recording of someone saying "Hello". Thus, Meme is a concrete class that has a set of media that all represent the same concept through specific presentation possibilities. When new kinds of media are added to the system, the Meme class does not change, and hence the assistant need not change either.

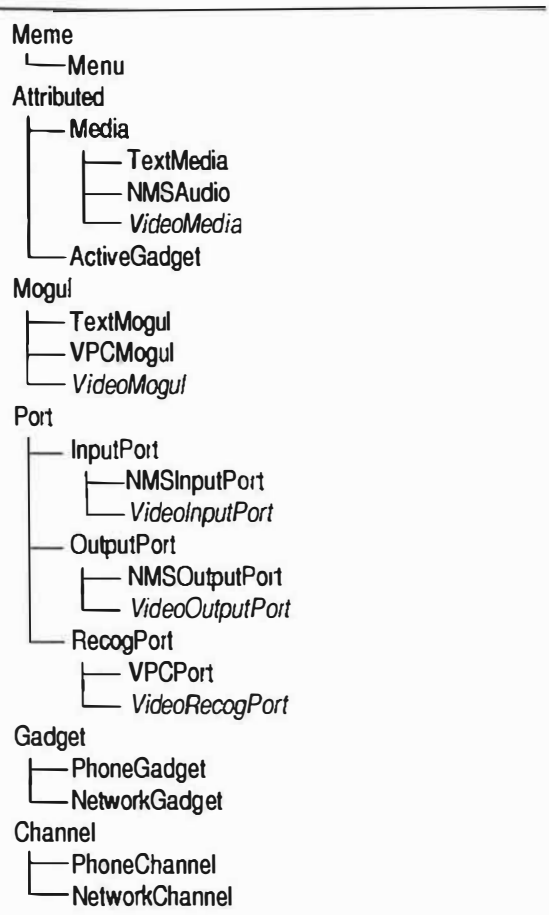


FIGURE 1. MMUI Class Hierarchy

Note that, of the two requirements on the media within a meme (that it be presentable and equivalent in meaning), the MMUI is concerned only with presentability, not meaning. It assumes that some human took responsibility for properly correlating the meme's contents to the meaning it is intended to represent.

All this is sufficient for output, but output is the simplest part of the system. Interpreting user choices gets more complicated, but builds on the same basic concept of grouping identical conceptual meanings that require different specific interactions into a single conceptual unit.

So we extend the basic Meme concept with a derived class that represents a set of choices. This is called a ***Menu***. It contains an ordered list of rows, each of which has a meme to represent its presentation and program-supplied data to be returned if it is selected. In order for it to be presented, and the user input interpreted, some expert code is required that understands how to use the data in the menu to recognize. This expert is called a ***Mogul***. A TextMogul would understand how to present the text media from each meme to the user as, say, a

numbered list and let the user type in the number of the choice they wanted. A voice-recognition Mogul will, as we shall see, be more complicated.

Presentation and recognition are done through an abstraction called an *ActiveGadget*. This represents an active one- or two-way channel between a process (the assistant) and a gadget (like a telephone) that is currently active, i.e., able to be interacted with (interacting with a telephone that does not have a person using it, and is thus not "active", is beyond the scope of this paper). Memes are presented on the channel, and recognition is done through it.

ActiveGadget and the base Media class are both derived from a base *Attributed* class that allows setting arbitrary name/value pairs on an object. The attributes on the ActiveGadget are used to store the output filtering preferences. The attributes on a Media object describe the attributes that particular media have. Matching between these values allows further refining of the list of Media that can be presented, once irrelevant Media (like voice media on a tty line) have been weeded out. This could be used to select verbose vs. non-verbose prompts, male vs. female speakers, etc. More on this later.

Before we talk about how this is translated into classes, objects, and other code-related realities, we will need a quick overview of the Wildfire architecture.

3. Wildfire Architecture

For the purposes of this paper, we will describe a simplified view of the Wildfire architecture from a MMUI-centric point of view. Basically, the Wildfire system is broken into a set of applications (of which the most visible is the assistant) that run on a specialized kernel called the Wildfire OS (WFOS for short). This exports functionality for communicating via ActiveGadgets to devices that support particular capabilities (such as speech recognition or recording).

The assistant gets an ActiveGadget by handing a Gadget description to the WFOS and getting back an ActiveGadget object that represents a channel of communication to that gadget. The assistant then requests capabilities of that ActiveGadget, which (if the addition is successful) will ensure that a port is attached to support that capability.

There are three major kinds of ports, all derived from an abstract *Port* class: *InputPort* for recording user input, *OutputPort* for presenting media, and *RecogPort* for ports that turn input into recognized selections. Currently a channel that represents normal communication with an assistant over a telephone line has the following ports attached to it:

- An InputPort that can record sound, such as a person leaving a message.
- An OutputPort to play pre-recorded sounds, such as system prompts.
- A speech recognition port that is derived from both OutputPort and RecogPort so that it can coordinate playing the sounds with setting up the recognition.
- A RecogPort to recognize touchtone selections.

As another example, a channel communicating to a user across a file descriptor currently has the following ports:

- An InputPort that records ASCII text.
- An OutputPort that prints ASCII text.
- A RecogPort that lets the user select an entry from a text menu.

4. And Now, Back to the MMUI

Now we can discuss in greater detail what the MMUI actually does.

First of all, let's examine what's in a Meme. A Meme has a set of Media-derived objects, each of which describes a representation of a single concept. Different media objects might be of the same C++ type, but still be logically distinct due to attribute differences. For example, the "Call Whom?" Meme might have all the following Media objects:

- A TextMedia object with the string "Call Whom?"
- A TextMedia object with the attribute "Length=Tutorial" and the string "Call Whom? Please say the name of one of your contacts."
- An NMSAudio[1] object with a pathname to a recording of a person saying "Call Whom?"
- An NMSAudio object with the attribute "Length=Tutorial" and a pathname to a recording of a person saying "Call Whom? Please say the name of one of your contacts."

There might be other recognized values for the Length attribute, such as "Brief" for which the representation might be "Whom?" or (for the audio) nothing at all. There might be other attributes, such as "Speaker=Female/Male" to allow the user to select a

1. NMS is the particular vendor whose board we use for basic audio presentation.

female or male voice for the prompts. For prompts that include the gender of a person (like "She's not here") that could be encoded in a "Subject=Male/Female" attribute.

There are a set of system memes, which are simply well-known memes within a particular name space. These are referenced through MemeID objects that uses the name of a meme to get a pointer to the actual meme from a table of system memes. These memes are not special in any other way beyond being entered in this name space. Many memes are created on the fly, such as user names, contact names, and message bodies.

4.1. Presentation

Now let us examine how the actual presentation of a Meme works. We will work on the example

```
ag << WfCallWhomP;
res = ag.prompt_response(WfOutCallM);
```

Memes are buffered until an explicit flush or a request for input. Thus, the `prompt_response()` invocation causes the `WfCallWhomP` prompt meme to be presented. Here is how that happens:

1. The `WfCallWhomP` MemeID is translated into a Meme reference.
2. ActiveGadget's `operator<<` method sends the Meme list (of one Meme in this case) to the WFOS to present through the channel.
3. The channel asks each attached OutputPort to present the Memes that have Media which that OutputPort understands.
4. Each OutputPort searches the Meme for Media objects of types that it understands. If it finds more than one such object, it does an attribute match to find the one whose attributes best match those set in the ActiveGadget. If there is a tie for best match, one of the best is picked randomly.[2]
5. When no more Memes are left to present, processing is complete.

When reducing the mass of possible Media down to the one that will be presented, the primary filtering is, of course, on presentable Media. For many Memes this may be enough. Attribute matching only matters if more than one presentable Media exists in that Meme. Attribute matching does not take adjacent Memes into account to smooth matching across a list of Memes (mostly because we've never found a use for it).

Ports find Media-derived objects that they understand using a runtime typing system. The NMSOutputPort will try and find Media that are at-least-a SimpleAudio media, which is a derived type that describes basic mulaw audio. The polymorphic behavior of Media and the various Port classes gives the MMUI great adaptive power. We will describe this in detail below, when we discuss how one would go about adding a new type of interaction to the system.

Also note that there is no 1-to-1 requirement between Ports and specific Media types. A Port may handle more than one kind of Media, and any number of Ports may understand a specific Media type. Since Ports are attached to channels because of requests for capabilities, it can be quite useful if a Port handles more than one Media, since it might reduce the number of Ports required to support a given capability.

Attribute matching is done in a very simplistic way. First, we take a list of the attributes meant to qualify the presentation, i.e., the attribute set from the ActiveGadget is overlaid with the set overriding the current presentation. (This can done with an option object analogous to the iostream manipulators.) For each attribute specified in the qualifying list, each presentable Media object is assigned a score: 2 for a match, 0 for a conflict, and 1 for a non-conflict (in other words, if the attribute in the qualifying list is absent from the Media, and hence, neither set correctly nor incorrectly). The presentable Media object with the highest score wins. This can mean that an object that directly conflicts with the qualifying list is presented, on the theory that the wrong output is better than no output at all. Wrong output tends to lead to complaints, which can often lead to getting the problem fixed. Silence tends to merely baffle.

4.2. Recognition

Recognition is somewhat more complex than presentation. This is partly due to the fact that more stuff has to go on in a two-way interaction than a one-way data dump, but voice menus also have an attribute rarely found in other menu systems: probability. When someone selects the third entry in a menu, that's what they've selected. When a speech recognition system tries to determine what you've selected, it can only return a *list* of probable selections. Its matching is only the best it can do. (The details of translating the actually output from the speech recognition algorithm into a probability is an interesting problem, but it is not in the scope of this paper; see [9,10,11].)

2. This is useful for areas where it is best to have the output vary. For example, the "You're welcome" Meme might have "You're welcome", "Don't mention it", "Happy to help", and so on, leading to a more personal interface.

This means that the output of a menu selection in a media-independent interaction is a list of candidate selections with some probability assigned to each. If the highest probability falls below some threshold, then the recognition must be considered a failure and handled appropriately. We notify the user in increasingly verbose ways that we didn't understand them, and then we finally give up on the whole command. (The issues surrounding the human factors of deciding how to handle these cases is another interesting, yet uncovered, topic; see [6,7,8].)

The menus can usually handle this kind of reprompting given the appropriate options (such as what to say when reprompting, and what level of probability is acceptable). There are, though, instances where the selection must be returned to a higher level.

Let's walk through the recognition request:

1. `WfOutCallM` is translated into a Menu reference. (There are system menus just like there are system memes, and the lookup is handled analogously.)
2. The menu, along with any specified options (none shown) are sent to the WFOS.
3. The WFOS waits for any asynchronous output request to finish.
4. It then asks each OutputPort to prompt/recognize on the pending output. This allows recognition ports to coordinate the ending of the prompt and the start of recognition.
5. The first RecogPort to say it is complete wins, i.e., its recognition result is the result of the overall recognition (no averaging or other inter-port data merge is done).
6. The responses are sent back to the MMUI, which asks the winning Mogul to translate the results into a canonical form, which is a list of ***MenuPick*** objects ranked by probability. (A MenuPick object holds the probability for that pick, an index into the Menu, and a pointer to the assistant-specified data associated with that row of the menu).
7. This list of MenuPicks is returned as the result. along with an overall confidence in the recognition, and (if appropriate) a failure status to distinguish failure due to timeout from failure due to unrecognizable noise.

If the recognition fails, step 6 instead consists of "bonking" the user with any other corrections necessary, all as specified by options provided to the menus via the ActiveGadget (which maintains the current default option settings) or specific overrides which can be given as an optional parameter to the `prompt_response()`.

Step 4 is where Moguls come into play. Speech recognition systems typically need a block of data that represents the entire menu of choices. This describes (in some device-dependent way) the differentiating aspects of the legal things a person can say. A common term for this is the ***vocabulary***. When you want to recognize words from a particular vocabulary, you must download the data into the device, and only then can you start recognition.

This means that a simple list of the possible choices is insufficient to represent a Menu on all input systems. While a list of Memes containing TextMedia would be sufficient to build a text or GUI menu on the fly, vocabulary building is a time-consuming process that offloads overhead from the recognition phase to a vocabulary building phase that must precede it. The Mogul was introduced to represent any overall menu-related information required by a RecogPort, such as vocabularies. We will deal more with how vocabularies are built below.

The coordination in step 4 is needed because, on some speech recognizers, if you do not coordinate the input and output, you can get very bad effects. On some systems, starting to recognize a person's voice while sound is still being played over the phone can lead to very bad effects, since you can start recognizing your own prompts as commands. On the other side, the prompt may have finished, but the recognition port may not yet be ready, i.e., the vocabulary may not have yet finished being downloaded to the device. Wildfire plays a little *blip* sound when it is ready to start recognition. This lets you know that, should the prompt finish too soon, you still need to wait. Otherwise people would start talking before the recognition window began, and Wildfire would start trying to understand them midway through their utterance. So this *blip* is coordinated with the output so that it does not play until the recognizing hardware is prepared.

We do not attempt to average results from multiple ports because we cannot see any meaningful way to do this in general. It is hard to even imagine that this can be usefully attempted. Imagine that the user had, when asked whom to call, said "Georgina Whit" and, before the requisite pause to signal the end of speaking, pushed the touchtone for "Never Mind" (the cancel command). How would one average such input?

One could imagine cases where combining various inputs you could increase the correctness of the recognition. On a video phone, for example, being able to match lip movements against what the speech recognition

thought was said might be able to help discrimination between possibilities. (I *did* say one could "imagine" such a thing.) To do this requires sophisticated interactions between the data available from multiple sources, and logically, within the MMUI, belongs in a single RecogPort that examines and correlates the relevant data. Neither the WFOS nor the MMUI could possibly broker this interaction in a general, abstract way applicable to other types of ports balancing multiple inputs.

4.3. Training

There are speech recognition systems that work on "speaker-independent" recognition. This means that any person should be recognized without having to train the system about how they personally speak. Although this is ideal in theory, even in the best current systems there are people with heavy accents or speech disabilities who cannot successfully be recognized.[3]

For this and other reasons, Wildfire (and hence, the MMUI) must support user-specific training for vocabularies. A large subsection of the MMUI is devoted to training, and the requirements of speech recognition training constrained parts of the design.

The MMUI supports a training call that sorts the user's menus based on what could use the most training, and presents them to the user one-at-a-time for training. To train a single menu, the user is asked to say each word.[4] Each mogul then uses that data to update its information.

Currently, only the speech recognition mogul VPCMogul[5] uses this step, but it is critical to its operation. The user's provided training for each word is added incrementally to the vocabulary so it can take effect immediately after the training is finished. At a later time, a batch process notices that there are new trainings for a menu, and it rebuilds its vocabularies in a more compact, and effective, form.

If you had, in some way, a recording of the user saying a word, you could add this without interacting with the user, but the vocabulary rebuilding work would still be required, which relies on access to the speech recognition device. This means that it is not possible to simply add a new meme to a menu, or to modify its contents by adding a new AudioMedia to change the recognition for that menu entry. Such a change requires active intervention by some code that understands how the device works. This is the Mogul's job. When a new row is added to a menu, each associated mogul is "introduced" to the new row's meme. This also happens when a row's meme is replaced; there is no mechanism for changing the contents of a meme in a menu except by wholesale replacement. It would clearly be possible to design a method to do so, but we have not yet needed to. And, obviously, when a row is deleted, all the menu's moguls are notified of that, too.

Currently, whenever a menu is created, all Mogul types are created and attached. There is a design, not yet implemented, to make this more dynamic based on content, but this wholesale approach has not yet proved to be a problem.

5. Extending the MMUI

One of the important design centers for the MMUI was the ability to extend the system by adding new kinds of interactions. There were several reasons for this requirement:

- Replacement hardware is constantly becoming available. If a new audio-playing board comes out that is much cheaper to use, it should be easy to change the system to use it, thus allowing a quick reduction in the price of shipping systems.
- New capabilities are coming quickly, too. The sophistication of speech recognition systems is increasing rapidly, and the MMUI should not be a bottleneck when deciding how quickly we can use better solutions.
- Completely new interactions should be quick to add. As two-way pagers become lighter and more widespread, we should be able to add them to the full system (for text-based ones) or for specific interactions (such as saying who is calling and asking if you'd like to take the call). The list of other possibilities is as long as your wired imagination can make it. Again, the MMUI side of this should not be the controlling factor in the schedule.

So we designed a system in which the code that must be added is isolated under five primary abstract base classes, Media, Mogul, Port, Gadget, and Channel.

3. Note that all discussions here pertain to speech recognition over the telephone. This introduces are large degree of variations due to noise, line quality, dropping, and so on. Voice recognition based on high-quality microphones directly connected to computers have an easier time of it.
4. Note that "word" here indicates a recognizable utterance that appears in a menu's vocabulary; in this sense, "tell me" is one word; the computer can't tell and doesn't care that it is, to humans, actually a phrase.
5. VPC is the particular vendor we use for most voice recognition systems.

- ***Media:*** If presentation will be done on the new device, a new Media will probably be needed to describe presentable data.
- ***Mogul:*** If recognition will be done on the new device, a new Mogul type will probably be needed to manage that complexity.
- ***Port:*** New input, output, and/or recognition ports will be required that understand how to work within the WFOS to drive the device.
- ***Gadget:*** Adding a new device may require adding a new Gadget to describe an address for the device
- ***Channel:*** A channel understands how to establish and terminate connections to Gadgets (e.g., how to dial the phone and hang up), and coordinate between multiple ports on a particular kind of gadget.

We have only briefly touched on the Gadget class. Gadget is an abstract class whose derived classes contain addresses that can be used to connect to specific targets, establishing a channel for an ActiveGadget. The details of that handshake are not terribly interesting here, but any device we talk to must be contacted at an address, and hence, must have a Gadget-derived type that contains that address. For a PhoneGadget the address is a phone number; for a NetworkGadget it is an internet address and port number that will be used for the network connection.

We have also not discussed Channels in any detail either, but they are rather straightforward. They are created to carry media to destinations specified by particular Gadget types. If the new device does not require a new Gadget type (e.g., it operates at a network address, for which Wildfire already has NetworkChannel type), it will not require a new Channel. If it does, the Channel will have to be able to resolve the address described in the Gadget to a Channel that can juggle the communication needs of that type of Gadget. Like Port, a new Channel is not very complicated beyond whatever is required by the device that connects to the machine. The NetworkChannel is trivial, since sockets are easy to manage. The PhoneChannel is more complicated because telephony has more intermediate and failure states.

Not all new systems will require all of these. Besides the obvious fact that output-only systems will not require recognition Moguls, new devices that work with phone lines for addresses would not require a new Gadget or Channel type. This is likely to be common; a fax system would need a new Media and Port type, but no new Gadget would be required.

To give a flavor of how we would add a new device to the MMUI, we will describe how to do this for a putative video-based system.

It is quite likely that the new video would be reachable either on the local network via an internet address, or over the phone like a video-conferencing system. Both of these Gadget and Channel types already exist in the Wildfire system, so we will just piggyback on them.

If a new address *was* required (some video systems, for example, require two phone lines to handle the volume of data), a new Gadget type would have to provide a way to hold such an address. A new channel type that recognized that address and knew how to establish a connection to the addressed video system would also have to be added. Since this is not the primary topic of the paper, we will skip the details of this mechanism, but suffice it to say that it is not very complicated, except whatever complication the video system itself may impose.

For this one needs a VideoMedia class, derived from the abstract Media class. The Media class doesn't require much of its derived classes; it mostly (being derived from the Attributed class) ensures that Media can be attributed. Almost all functionality is added in the specialized classes. The VideoMedia would presumably store a pathname of a file that contained the media, and probably a start and end frame within the file.

Providing input and output ports for the video system is again relatively straightforward. New classes would be derived from the InputPort and OutputPort classes that overrode the pure virtual `record()` and `present()` methods respectively. The `record()` method's main job would be to dump the video into an appropriate place, and create a VideoMedia object that described it.

The `present()` method would paw through the Media in the list of pending output, looking for VideoMedia objects. If it found one, it would do the same for the next meme in the list, continuing on until it reached either the end of the meme list, or a meme that didn't contain any VideoMedia. It would then peel off the memes it *could* reasonably present, and, for each one, find the best attribute matched VideoMedia and present it.

As it currently stands, we could use the system described only for input and output, but not for recognition. We could take a video message, or play a video clip, but we couldn't ask any questions. Let us presume that this video system has an attached keyboard for answering questions (since I suspect a sign-language gesture recognition system is currently a tad beyond even the limits of handwaveware).

It is possible even here that we can avoid any hard work, since there already is a text recognition port, and if the video system understands simple ASCII I/O, a TextRecogPort would fulfill our engineering needs. It would not, however, fill our pedantic needs, so we will assume that this is not so.

So we need to add a VideoRecogPort and a VideoMogul. The Mogul class requires its derived classes to handle notification of changes in the Memes of the Menu, and to be able to canonicalize the list of MenuPicks into a form presentable to the application. The VideoMogul would be interested in the TextMedia of the menu so it could create a menu on demand (any Mogul can look at any Media to do its work). It might also be interested in VideoMedia that presented the choices in a video form, should the user want some help understanding the available choices.

The VideoRecogPort would present the list of choices as text and allow the user to select one in some way.

Notice that this does *not* preclude doing simultaneous speech recognition on the video system's microphones. Just as touchtones and voice can coexist on a single ActiveGadget, so can voice and the video systems text selection mechanism (and touchtones too, if desired). Whichever got an answer first would govern any particular recognition, but each recognition could be responded to in any available way.

6. Current State

The current state of the MMUI allows a linear presentation style with quite a lot of separation between the internals of the code and the particular media presentations required to present, record, or recognize in a particular case. However, as currently implemented, the MMUI still has major weaknesses if one is to consider it as a general interface abstraction.

First, it has no concept of "sentence". The presentation of Memes is linear in nature, with the order specified by the assistant. This is one problem (of several) that makes porting the assistant to a different language difficult. The natural or allowed order of presentation could be quite different.

There is also no concept of "dialog". Most actions require more than one piece of data. Again, different languages may impose a different expected or required order on gathering the data. For example, in English it is natural to say "Call Gordwina at work", but another language may prefer "Call the workplace of Gordwina". Further, a particular piece of data may affect the valid values of another one; Gordwina may or may not *have* a work phone.

There is also the issue of context. In a conversation, much data can be left out, inferred by the context. Overall context (for example, the subject of discourse) is easily built into the assistant interactions, since it is obvious and shared — we are talking about calling people, for instance, by the nature of the dialog, and so the dialog designer can make the interface understand that context.

But specific context is harder to provide. It would be nice to use words like "it" or "them". But without a notion of the type of legal referent, and the history of previous interactions that might state or imply a referent of that type, such words can only be used in a highly constrained way.

These problems limit the use of the MMUI to linearly presented interfaces. If one wanted, for example, to add a desktop GUI interface, one could only do so using a series of single-question prompt-response dialogs. Graphical icons could be included as part of the dialog description media, but it would not be possible to pop up a full "Place A Call" dialog that let one specify both whom to call and where to call them.

Designs exist to address these problems by adding classes and protocols to represent each, but as yet no prototyping has been done to try and prove them actually useful in the cauldron of the real world.

On the other hand, the Wildfire system is currently available, and is built using this infrastructure. It is quite possible to interact with the assistant using intermixed voice and touchtone commands, to listen to data recorded in different formats (hence, having different Media objects representing them), and to use attributes to select particular media (tutorial vs. standard prompts). As an experiment, we added a new channel type able to talk across a text-based two-way pager. With no modification to the assistant, it was possible to see the prompts and respond via the pager's keyboard just as one did when communicating via a Network Gadget to a local terminal emulator.

The MMUI has proven to be a useful tool in isolating many details of the interface presentation from the duties of the assistant. It has proven its adaptability to different flavors of linear presentation. It is easy to add new types of presentation and recognition, and should be possible to extend to provide greater isolation to the presenter. These benefits make it a useful abstraction in designing media-independent interfaces that must be presented in a media-dependent world.

Acknowledgments

Tony Lovell, Vinnie Shelton, Keith Gabryelski, Rich Miner, Greg Cockroft, and Dave Pelland contributed significantly to the design and implementation of the ideas presented here. Bill Warner created the concept of the Wildfire Assistant which motivated this design.

References

[1] Eguene Cicarelli, "Presentation Based User Interfaces", Thesis, MIT AI Lab, Technical Report AI-TR-794, 1984

[2] Pedro Szekely, "Modular Implementations of Presentations", Proceedings SIGCHI+GI 1987, pp. 253-240.

[3] Pedro Szekely, "Separating the User Interface from the Functionality of Application Programs", Thesis, ECMU 1988.

[4] Scott McKay, William York, Michael McMahon, "A Presentation Manager Based on Application Semantics", Proceedings SIGGRAPH Symposium on User Interface Software and Technology 1989, pp. 141-148.

[5] H. Rex Hartson, Deborah Hix, "Human-Computer Interface Developments: Concepts and Systems", ACM Computing Surveys, 21:1, pp.5-92.

[6] Candace Kamm, "User Interfaces for Voice Applications" *Voice Communication Between Humans and Machines*, National Academy Press, Washington, D.C., 1994.

[7] Eric Ly, Chris Schmandt, "Chatter: A Conversational Learning Speech Interface", *AAAI Spring Symposium on Intelligent Multi-Media Multi-Modal Systems*. Stanford, CA, March 1994.

[8] Nicole Yankelovich, Gina-Anne Levow, Matt Marx, "Designing SpeechActs: Issues in Speech User Interfaces", Proceedings, SIGCHI '95 Conference on Human Factors in Computing Systems.

[9] Gordon E. Pelton, *Voice Processing*, McGraw-Hill, 1993.

[10] L. R. Rabiner, B. H. Juang, *Digital Processing of Speech Signals*, Prentice-Hall, 1978.

[11] Kai-Fu Lee, *Automatic Speech Recognition*, Kluwer Academic Publishers, 1989.